School of Computation, Information and Technology - Informatics

Technical University of Munich

Bachelor's Thesis in Informatics

Load Balancing on Dynamically Adaptive Grids in ExaHyPE

Jörn Dominik Fischbach

School of Computation, Information and Technology - Informatics

Technical University of Munich

Bachelor's Thesis in Informatics

Load Balancing on Dynamically Adaptive Grids in ExaHyPE

Lastbalancierung auf dynamisch adaptiven Gittern in ExaHyPE

Author:	Jörn Dominik Fischbach
Supervisor:	Prof. Dr. Michael Bader
Advisor:	Mario Wille, M.Sc.
Date:	30.01.2023

I confirm that this bachelor's thesis is my own work and I have documented all sources and material used.

Munich, 30.01.2023 Jörn Dominik Fischbach

Abstract

This work revolves around dynamic workloads on distributed systems. Simulations that require finer resolutions in subdomains of interest induce load imbalances that cannot be rectified with static load-balancing. This work aims to analyze available dynamic load-balancing strategies of ExaHyPE 2. ExaHyPE 2 is an HPC simulation software that depends on Peano 4, a dynamically adaptive grid traversal framework. We deduced and justified custom load-balancing metrics for ExaHyPE 2. These metrics can be easily tailored for analyzing other applications experiencing MPI+X scaling problems. The metrics are then used for evaluating previously analyzed strategies. During testing, problematic load-balancing behavior is highlighted. Intra- and inter-rank metrics give insights into load distribution and density. Conclusions about the quality of the load-balancing strategies were drawn with multiple load-balancing algorithms yielding deficient results. Finally, a fine-granular post-processing tool for plotting workload over time is provided that will aid future work with analyzing dynamic load-balancing statistics of ExaHyPE 2 and Peano 4.

Contents

1. Introduction and Background

In high-performance applications, the challenge of scalability is inevitable. In the field of high-performance computing (HPC), parallelism is crucial. Programs that are not capable of scaling with available hardware will be inferior to capable parallel applications. Load-balancing is relevant in HPC applications in the context of multiple threads that are scheduled on a single processing core, as well as multiple processes that are mapped onto different processors. When exascale supercomputers run resource-expensive simulations, uneven load-balancing bottlenecks programs, although the hardware could achieve faster data processing. The framework ExaHyPE 2 is an exascale hyperbolic partial differential equation (PDE) engine for PDEs in the first-order formulation. The uneven load-balancing in ExaHyPE 2 induced by dynamically adaptive mesh refinement prevents applications from running efficiently on modern supercomputers. It restricts the exhaustion of available computing resources.

Initially, load-balancing algorithms can be classified by multiple properties. In the course of this thesis, two main properties are most relevant. First, an algorithm implements a static or dynamic load-balancing based on its degree of adaptivity. Second, it implements centralized or distributed load-balancing depending on the number of load-balancing instances that make decisions simultaneously [1]. The difference between static and dynamic algorithms is the following:

- **Static**: Makes decisions independent of the run-time state of the application to balance. (Stateless load-balancing)
- **Dynamic**: Makes decisions based on the state of the application and its gathered run-time information. (Stateful load-balancing)

The difference between centralized and distributed algorithms is explained below:

- **Centralized**: Makes decisions on a singular master program instance or a singular master thread of a process. (Master decides, workers comply)
- **Distributed**: Makes decisions on all program instances or all process threads. (Equality in decision-making)

In the application that this thesis targets, ExaHyPE 2, dynamic load-balancing is the only option for dynamically changing workloads. This is based on a property of Peano 4[1], the underlying dynamically adaptive Cartesian mesh traversal framework. As the mesh of Peano 4 is subdivided into chunks, which are then balanced, Peano 4's feature to refine or coarsen the simulation mesh during the program's run-time induces differing workloads of chunks traced over time. Therefore, a static load-balancing algorithm like a pure round-robin or

[1]Peano 4 and ExaHyPE 2 source code: https://gitlab.lrz.de/hpcsoftware/Peano
Peano 4 and ExaHyPE 2 documentation: https://hpcsoftware.pages.gitlab.lrz.de/Peano/html/index.html

randomized algorithm is not viable, as they do not account for the differing workloads and induce broad stalling. A dynamic load-balancing algorithm is a necessity for this software. Whether a centralized or distributed load-balancing approach achieves better results will be evaluated in this thesis.
The software that ExaHyPE 2 was built on, Peano 4, has various features and dedicated, informative papers about its design decisions and stack-based memory use [2, 3]. Peano 4 uses spacetrees, a generalization of octrees, in order to partition the space domain into a multilevel grid that provides flexible precision. This flexible precision provides custom resolution for areas of interest during a simulation. Peano 4 is a tree-based adaptive mesh refinement software. The overall 13 design decisions that are discussed in the original paper reveal illuminating details about Peano 4's internal structure and data management. In the following, the design decisions relevant to this thesis are summarized and briefly explained to facilitate an entry into the topic. Design decisions that, for example, target user-injected code for applications are not discussed.

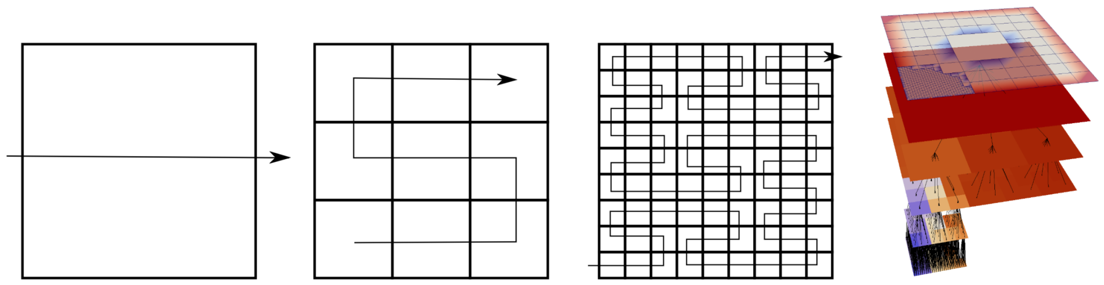
Figure 1.1.: Peano 4's space domain decomposition of a 2D grid. [2]

The first design decision denotes that uniqueness is conferred upon cells and vertices of spacetrees by their spatial position and corresponding level. Consequently, while multiple vertices may spatially coincide, they reside at different levels [2]. This distinction of levels can also be viewed in Figure 1.1, where the first three grids depict the same space that was subdivided recursively. The level of the cells viewed from the leftmost to the third, rightmost grid differs by two, although they exist in the same space.

Second, the Peano 4 software tripartitions space along each dimension as the Peano Space-Filling Curve (SFC) is used for linearizing the space [2]. This partitioning can also be examined in the two-dimensional grid of Figure 1.1. For dimension 3, the SFC can be extended to fill a cube; for dimensions greater than 3, the Peano curve can linearize hypercubes.

Third, the grid traversal order is prescribed by Peano 4, i.e., all tree vertices are traversed but the user cannot influence the traversal order. Nonetheless, assured temporal constraints establish a guaranteed partial order on the traversal's transitions [2].

Fourth, Peano 4 provides strict element-wise multilevel data access by default. However, the relaxation of these data access permissions is permitted. This is achieved by allowing each vertex to point to its adjacent cell data at the cost of 2^d pointers per vertex, where d indicates the dimension [2].

Fifth, Peano 4 utilizes the Peano SFC to linearize the tree data structure. It adheres to a Depth-First Search (*DFS*), causing the traversal to interpret the spacetree as a bit stream [2].

Using the DFS, traversing through different grid levels is prioritized over traversing to neighbors with the same parent vertex.

The final crucial design decision, which determines the spatial domain's decomposition and whether data is replicated to preserve the spacetree topology, leads to Peano utilizing a non-replicating data layout and implementing top-down tree partitioning. This establishes a logical master-worker topology on the ranks. [2]. More in-depth information on the decomposition and splitting is provided in the course of this thesis.

After these insights into the Peano 4 software, the hybrid memory management is especially noteworthy. Peano 4 provides flags for its hybrid memory management that utilizes shared and distributed memory to scale on high-performance systems. For shared memory, Peano 4 provides multiple library support like OpenMP, Intel® oneAPI Threading Building Blocks, SYCL, and more. MPI is used for distributed memory management. The goal of this hybrid MPI+X approach is to minimize inter-node communication by roughly subdividing the space onto compute nodes and then letting each node exploit the shared memory for quick access to, e.g., adjacency information, as most of the memory for tree traversals is held locally.

The main difficulty of this approach is compensating for dynamically changing workloads as the multilevel grid refines or coarsens during run-time. As dynamically changing loads in applications are common in HPC, this thesis is helpful to ExaHyPE 2, and its core ideas are of interest to other HPC applications facing a similar challenge. For this reason, the final goal of this thesis is to analyze and evaluate available load-balancing strategies of Peano 4 and lay the basis for a new superior strategy that unites the advantages of the analyzed strategies.

2. Related Work

This thesis builds on Peano 4, an open-source framework that uses automatons to apply solvers on dynamically adaptive Cartesian meshes. To summarize Peano 4's structure roughly, the mesh's adaptivity is achieved by hosting the space domain, the Cartesian grid, on multiple spacetrees. First, a single spacetree covers the entire domain. This spacetree can refine the underlying Cartesian grid by increasing its depth and hosting new, finer grids (cf. Figure 1.1). One of these grids replaces a cell, a leaf node of the previous spacetree. This spacetree, or the entire space domain, can be traversed in parallel by splitting the spacetree into multiple single spacetrees. The tree data structure allows splitting off an inner node and hosting a new spacetree from this node as its root. For a more in-depth explanation, refer to the paper that compares and evaluates different design decisions made for the software Peano 4 [2]. Almost all code that forms the core of Peano 4 is written in C++, whereas setup, post-processing, visualization, and other features are realized in Python.

As Peano 4 provides the back-end of a simulation, the middle-ware could be a self-written framework or, in the case of this thesis, ExaHyPE 2, *An Exascale Hyperbolic PDE Engine*, view [4, 5]. The use cases of such an engine are various but primarily focus on simulating gravitational waves in astrophysics simulations or seismic waves, e.g., earthquakes. The frontend of a simulation consists of the user-specific application code. ExaHyPE 2 builds on the spacetree management logic of the Peano 4 framework and provides additional functionality for specifying partial differential equations and, e.g., PDE terms, boundary conditions, solvers, and load-balancing strategy for the simulation.

Besides the ExaHyPE 2 joined with Peano 4, the field of tree-based adaptive mesh refinement is the subject of current research. In the context of balancing loads for these scenarios, the field of graph partitioning is relevant. The problem of partitioning a graph is proven to be NP hard [6], but heuristics that can yield satisfying results have been developed in the past. While graph partitioning is a subject of different kinds of research, the most frequent subject is solving optimization problems. In order to balance loads perfectly across p processes with t threads, a $(p \cdot t)$-way graph repartition is searched for. As Peano 4's code base is complex, memory management and memory locality are important, so repartitioning Peano 4's internally decomposed spacetree demands further research. After repartitioning, the *divide-and-conquer* approach enables the exploitation of all parallel computing resources by simply assigning each thread one partition. The decision characteristic of how the graphs should be repartitioned is the weight of the vertices.

For future developments on Peano 4, an insight into the *ParMETIS* library may serve useful [7]. ParMETIS follows a multilevel strategy to repartition graphs by first coarsening the existing graph, then partitioning the coarse graph and finally refine the partitioned graph again. It should be noted that ParMETIS' development has come to a halt; the last maintenance of the code is dated March 2023.

When reviewing the history of the thesis' main problem, balancing loads of structured adaptive mesh refinement software led to initial research in 2001. The paper introduced a load-balancing algorithm that improved early load-balancing efforts by slicing workload into smaller sub-workloads and distributing these across processors [8]. The introduced load-balancing algorithm does not take memory locality into account, resulting in unnecessarily dispersed load redistribution. Peano 4 tries to keep the memory as local as possible, so an approach similar to the early beginning of the problem is suboptimal. Interestingly, the paper also introduced metrics for measuring the load-balancing quality. In the course of this thesis, custom load-balancing metrics for measuring the quality of load-balancing will be developed and tailored to the set-of-trees data structure of Peano 4. The metrics of the paper can be considered as a reference for the self-developed metrics.

The load-balancing optimization of MPI+X applications is also the subject of very recent research. MPI+X denotes general hybrid memory using systems, with MPI for inter-process communication and a variable X for intra-process data exchange. A research group focused on how to improve performance in an MPI+X environment. Their key idea was to postpone the work induced by the inner cells of an adaptive mesh and prioritize the cells at either MPI boundaries or AMR-relevant locations [9]. This prioritized cell processing speeds up MPI communication as the lower priority computations can be performed during the MPI communication of already finished high-priority cells. The realization of these low- and high-priority computations implements a task-based approach. During mesh traversal, ready tasks are spawned directly and no task-dependency graph is explicitly built. The non-prioritized tasks form so-called enclaves, tasks whose low priority allows computation in the background.

In general, load-balancing approaches for distributed systems can be categorized. This categorization differs static load-balancing (SLB) from dynamic-load-balancing (DLB) and further subdivides DLB into the categories centralized or distributed [10]. This categorization served as an entry point to the dynamic load-balancing field and was useful in outlining the structure of this thesis. Therefore, the following chapter revolves around the load-balancing that is available in Peano 4 and thus also in ExaHyPE 2, its categorization and implementation.

3. Load-Balancing in Peano 4

Peano 4's current load-balancing strategies are stateful and decentralized or stateful and centralized. The stateful characteristic of the strategies is inevitable for a well-balanced multi-core and multi-threaded program, see Chapter 1. As Peano 4 follows a single program, multiple data (SPMD) programming model, every process or MPI-rank hosts an instance of the load-balancer, which can make internal decisions about its local threads and spacetrees, but also invoke inter-rank rebalancing if the own rank becomes too work-heavy or light-work compared to other ranks. This SPMD property of Peano 4 forces load-balancers to work decentralized, refer Chapter 1, except for strategies with the *spread-out* prefix as they emulate decentralized behavior by rank-wise querying the rank ID and, e.g., stagnating or waiting if the rank ID is not the master rank's ID of 0.

Some core concepts are especially noteworthy to facilitate understanding interactions during the load-balancing processes. The spacetree set $\mathcal{T}_i$, a singleton, holds the local trees of the i-th rank p_i. Additionally, there is a blacklisting functionality per rank which can veto tree splits. This is done by adding trees into a blacklist map and frequently updating this blacklist $\boldsymbol{B_i}$. Blacklisted trees of rank i are kept on the blacklist B_i for three grid traversals by default. This behavior stems from the tree-splitting process of Peano 4's core routines, where a tree is only marked to split in the first grid traversal, then in the state of splitting in a second traversal, and finally split with ownership transfer in the final, third, grid traversal. This blacklisting ensures the integrity of trees and prevents forking off trees that are already in the splitting process.

Also, the load-balancing statistics, which are used to make decisions, are not generated on demand but rather stored and updated throughout the run-time by the load-balancer itself. To determine the load of a spacetree, the local unrefined cell count is the default for the load-balancing toolbox of Peano 4, but the possibility of implementing custom metrics for the load determination of trees and processes is also provided.

3.1. Overview of Key Load-Balancing Stages

In Peano 4, all load-balancing strategies run through a set of internal states. The current state is an essential internal variable determining the further progression of load-balancing actions. Peano 4's load-balancer can be interpreted as a stateful automaton $\boldsymbol{L}$. By the SPMD model, this automaton is an instance of a process and can be indexed as $\boldsymbol{L}_i$ for process i. During run-time, all load-balancers have an identical underlying implementation. However, they may be in different internal states at the same real-time depending on the inter- and intra-rank statistics like local unrefined cells, recent local tree splits, and more. The list of states on which all current load-balancing strategies rely is:

Undefined Default, initial error state of `AbstractLoadBalancing` class. Unused by all other load-balancing strategies.

InterRankDistribution The code is in the earliest load-balancing stage. The load shall be distributed among ranks; intra-rank distribution is not relevant yet. All load-balancing strategies start in this state and prioritize the distribution of inter-rank workload.

IntraRankDistribution At the beginning of this state, there's at most one tree per rank. The code has spread over all ranks. Next, spread over all threads inside the ranks.

InterRankBalancing The initial spread is completed. In this state, the load is balanced between the ranks to meet the load-balancing quality.

IntraRankBalancing The inter-rank load-balancing quality is sufficient. Now, balance in the ranks, i.e., among local spacetrees.

WaitForRoundRobinToken The current rank waits for a token to perform inter-rank balance changes for which other ranks comply in a round-robin fashion.

Stagnation The code has achieved a sufficient load-balancing quality and does not balance further. It is not switched off, as switched-off states can transition to active states again.

SwitchedOff The code has locally switched-off the load-balancing. It may be reactivated but currently satisfies expectations or needs to be inactive so as not to interfere with, e.g., inter-rank load-balancing decisions of a different rank.

Some additional load-balancing states for specific strategies will be noted if and only if necessary.

3.2. How Trees are Split in Peano 4

This section explains how splitting off cells is implemented in Peano 4. First and foremost, the splitting is one of the core features of Peano 4 and, therefore, has an extremely high code complexity. Thus, this explanation is restricted to general information regarding the splitting. An in-depth analysis exceeds the scope of this thesis. The splitting is explained extensively in publications of Tobias Weinzierl, refer to Chapter 7 of [2] and Chapter 5 of [11]. Spacetrees and the spacetree sets that allow parallelism are highly relevant for explaining the splitting feature, the heart of Peano 4's tree logic. Additionally a load-balancer needs to call the `split(...)` function:

```
peano4::parallel::SpacetreeSet::getInstance().split(
    sourceTree, // Source tree ID from which the cells will split off
    peano4::SplitInstruction{numberOfCells,
        peano4::SplitInstruction::Mode::AggressiveTopDown},
    targetRank // Target rank ID, set own rank for intra-rank split.
);
```

Listing 3.1: Call from load-balancer to `SpacetreeSet`

The second argument, a split instruction, contains the number of fine grid cells that shall split off as first and the mode to split off as a second value. This `triggerSplit(...)` call to the spacetree set initiates preparation for the split. Currently, Peano 4 provides two splitting modes: the aggressive top-down and finer bottom-up approach. Internally, Peano 4 uses top-down tree splitting to keep the spacetree topology consistent. The leitmotiv is deploying new trees to ranks by cutting out subtrees of the initial spacetree. As the space-filling Peano curve implies an order over all spacetree cells, cutting out a subtree and deploying it to another rank keeps the total order consistent as long as the remote root is logically connected to its parent. Independent from the internal storage and tree management, the two different splitting modes determine how to find the ideal root note from which to split. In both modes, the spacetree set, after reserving a spacetree ID for the new tree and resolving the spacetree described by the passed `sourceTree` ID, delegates the split to the tree by calling `tree.split(newID,instruction)`, with `tree` being the resolved tree that shall split off cells. Internally, the tree uses a three-stage splitting realization. First, cells are marked to split in the initial iteration. Then, they are in the process of splitting after the second iteration. Finally, they are considered split in the third, i.e., the last iteration. For the bottom-up splitting, the precise number of finest grid cells that are split off is guaranteed to match the requested number of cells to split as best as possible. For the top-down splitting, a number equal to or greater than the requested cells that are split off is sufficient and will be used for splitting. Peano 4 tries to stay as close to the requested number as possible, but constraints that are elaborated on in the Peano 4 documentation under the topic *Domain Decomposition* have priority over requested cell counts. All load-balancing will have to decide which splitting mode is most suitable. The advantage of top-down splitting is that the probability of splitting off trees as a whole is maximized. The advantage of bottom-up splitting is the best chance of meeting the requested number of cells to split off, at the risk of fragmenting the tree-topology quicker. After briefly explaining the function stack that the load-balancer uses to instantiate new trees and the main difference between bottom-up and top-down splitting, we move on to the load-balancing strategies that Peano 4 currently provides.

3.3. Available Load-Balancing Strategies

This section serves as an entry point to understand already existing load-balancing strategies. As a base for all load-balancing schemes of Peano 4, an abstract load-balancing class exists that offers wrappers with semantic checks and core functionality to retrieve grid and rank statistics. In the case of writing one's own load-balancer for Peano 4 or comparable MPI+X applications, the abstracted functionality is substantial. Some functions that mostly implement variable-value-getting logic or simple checks are omitted in the following to focus on the essence of the load-balancing strategies.

Additionally, the input parameters for the below-defined functions may vary from the actual implementation in ExaHyPE 2. This difference is because, e.g., an initial load-balancing configuration created with user parameters at the start of the application is realized as member variables of the load-balancer in ExaHyPE 2. In all algorithms of this thesis, member variables are explicitly named as input for functions to avoid unexplained variable names and not require any previous knowledge. Usually, the two most relevant functions that update the internal load-balancing state and take load-balancing action, i.e., initiate splits, rely solely on member variables and no passed arguments.

At present, five unique load-balancing strategies have been implemented: bi-partitioning recursively, splitting an oversized tree, spreading out, spreading out hierarchically, and spreading out once the grid stagnates. Additionally, there is the possibility to cascade these strategies, i.e., running through a hierarchy of load-balancing strategies, switching to the next when the previous strategy stagnates. The following subsections explain the core idea of each strategy and summarize their behavior for state-switching and applying the load-balancing updates.

3.3.1. Recursive Bi-Partition

The first and most straightforward strategy already implemented is the recursive bi-partition. Its core approach to achieve better load-balancing quality is splitting the trees up aggressively. The heaviest local tree w.r.t the cost metric, which defaults to the cell count, is determined. Successively, some checks, like the tree not being blacklisted, are performed. Then, the recursive bi-partition strategy ideally splits the tree into two equal-sized trees. For this, a top-down splitting scheme is used (cf. Section 3.2).

Algorithm 1: `updateState` of recursive bi-partition

```
Input:  state, roundRobinToken
Output: state, roundRobinToken
1  Function updateState(state,roundRobinToken):
2      if state = "SwitchedOff" then
           // do not update
3      else if areRanksUnemployed() then
4          state ⟵ "InterRankDistribution"
5      else if isInterRankBalancingBad() and isMyTurn(roundRobinToken) then
6          state ⟵ "InterRankBalancing"
7      else if isIntraRankBalancingBad() then
8          state ⟵ "IntraRankBalancing"
9      else
10         state ⟵ "WaitForRoundRobinToken"
11         roundRobinToken ⟵ (roundRobinToken + 1) % getNumberOfRanks()
12     return state, roundRobinToken
```

Algorithm 1 shows that the functions in the conditions of the if-clauses do not use the previous internal state, nor is it used to switch the state in the Algorithm 1 itself. The independence from the state means that the current state is almost exclusively unused to update, i.e., the state-transition graph is a directed, fully connected graph for the states of inter-rank distribution, inter-rank balancing, intra-rank balancing, and waiting for the round-robin token. The exception is the node of the switched-off state, which is fully connected to the other states with incoming edges, but the only outgoing edge is a loop to itself. Before investigating the splits instructed by this load-balancing strategy, some functionality needed to be extracted, refer to Algorithm 2.

Algorithm 2: Utility function for recursive bi-partition

```
Input:   heaviestSpacetree, state,
         numLocalUnrefCellsOfHeavSpacetree
Output:  boolean

Function canSplitRB(heaviestSpacetree,numLocalUnrefCellsOfHeavSpacetree,
                    state):
    return fitsIntoMemory(state) and heaviestSpacetree $\neq -1$
           and not isBlacklisted(heaviestSpacetree) and
           numLocalUnrefCellsOfHeavSpacetree $>$ getMinAllowedTreeSize(state)
```

The function `canSplitRB()` shortens repeated conditional checks for the recursive bi-partition strategy, as these condition checks are performed before triggering any split. They confirm that, from the perspective of the load-balancer, nothing interferes with instructing a split on the tree of concern. The return statement involves the evaluation of multiple conditions. First, `fitsIntoMemory(...)` uses a worst-case estimate for the heaviest local spacetree's splitting behavior and ensures sufficient memory. Second, the heaviest local spacetree passed into the function must exist, i.e., the rank is not empty. Third, the spacetree must not be blacklisted, i.e., it is not already splitting or performing other AMR operations, which could interfere with the spacetree topology's consistency when trying to split. Last, the number of local unrefined cells of the heaviest local spacetree must exceed the minimum allowed tree size, a command line option that can be passed into the application. Splitting off a tree is forbidden if the tree size is the prescribed minimum. If all these conditions hold, a split will be instructed. After clarifying this functionality, view Algorithm 3 for the decision-making on splitting off cells.

Initially, four values are retrieved and calculated in Algorithm 3. The variable assignments are drawn in front of the switch-case expression as it serves better readability. For implementation, the variables can be held in their local scopes of need. The first two values are the tree IDs of the heaviest local spacetree and a spacetree within the tolerance to the heaviest local spacetree[1]. The justification for both values is that during inter-rank distribution, the code only allows one of a range of spacetrees for a split if they are within a tolerance interval under the heaviest local spacetree. During the inter-rank balancing stage of recursive bi-partition, this freedom of choosing a sufficiently good tree for the split is removed.

Only the heaviest local spacetree is considered for the inter-rank balancing. The number of local unrefined cells of the heaviest local spacetree is assigned to the third variable. This number is directly reused to calculate the cells that each core should receive. The naming convention may be misleading. For clarification, `cellsPerCore` describes the cells that go to another rank's new tree (case *InterRankDistribution*), another or the own rank's new tree (case *InterRankBalancing*), or just the own rank's new tree (case *IntraRankBalancing*). Furthermore, if the tree is larger than the configured minimum size of any tree, the previously

[1]See Appendix Section A.1, Peano 4 source code is inconsistent with strategy

Algorithm 3: `updateLoadbalancing` of recursive bi-partition strategy

```
Input:  state, config
Result: triggering a tree-split
Function updateLoadbalancing(state, config):
    heaviestSpacetreeTol ⟵ getIdOfHeaviestLocalSpacetreeInTolerance()
    heaviestSpacetree ⟵ getIdOfHeaviestLocalSpacetree()
    numLocalUnrefCellsOfHeavSpacetree ⟵ getWeightOfHeavLocSpacetree()
    cellsPerCore ⟵ max(1, numLocalUnrefCellsOfHeavSpacetree /2,
                        config.getMinTreeSize(state))
    switch state do
        case "InterRankDistribution" do
            if canSplitRB(heaviestSpacetreeTol,
                          numLocalUnrefCellsOfHeavSpacetree, state)
            and getLightestRank() ≠ getMyRank() then
                triggerSplit(heaviestSpacetreeTol,cellsPerCore,
                             getLightestRank())
        case "InterRankBalancing" do
            if canSplitRB(heaviestSpacetree,
                          numLocalUnrefCellsOfHeavSpacetree, state) then
                triggerSplit(heaviestSpacetree,cellsPerCore,
                             costMetric.getLightestRank())
        case "IntraRankBalancing" do
            if canSplitRB(heaviestSpacetree,numLocalUnrefCellsOfHeavSpacetree,
                          state) then
                triggerSplit(heaviestSpacetree,cellsPerCore,getMyRank())
        otherwise do
            // nothing
```

heaviest tree is split in half, $\lfloor \textit{number of local unrefined cells of heaviest spacetree}/2 \rfloor$, i.e., bi-partitioned. **Note**, the *triggerSplit(...)* function of the recursive bi-partition is just a wrapper for the previously mentioned `split()` function of Listing 3.1 with blacklisting and statistic updating.

When this concept is applied repeatedly, and the already split trees are split again with the same rules, the domain decomposition is recursive. This is the core concept of the strategy. When reflecting on the bi-partitioning of trees, this principle can lead to bad results if applied at the start of an ExaHyPE 2 application. The scheme assumes a fast refining and building grid and thus splits aggressively. Although the concept is functional, the logarithmically decreasing tree size of the strategy leads to large size differences in the early stages of grid construction and imposes imbalances that have a grave impact due to the further refinement of the grid during construction. Its delayed use is recommended.

3.3.2. Split-Oversized-Tree

The split-oversized-tree load-balancing strategy has the core idea of not letting trees exceed a maximum cost threshold. The strategy uses the `doesLocalTreeViolateThreshold()` function, see Algorithm 4, to determine whether at least one of the local spacetrees violates the preset threshold. The threshold function calls `getTargetTreeCost()`, which in turn uses a minimum over the per-rank thread count, the predefined maximum number of spacetrees per rank, and the current number of local spacetrees to determine whether any of the local spacetrees are oversized. After retrieving this minimum, the global cost, i.e., the summed-up cost of all ranks and their trees, is divided by the rank count and the above-mentioned minimum to obtain the tree cost threshold. The threshold is pessimistic, as dividing by the minimum maximizes the quotient. Let the following example clarify how the threshold is computed:

Let the number of processes, i.e., ranks, be eight, with three spacetrees currently hosted on each of the eight ranks. The maximum number of trees per rank is set to four. The threads per rank are set to six. The local costs for the trees are 200, 400, and 500 for the rank that currently decides whether to rebalance. We assume the global costs are 7200. First, we perform the aforementioned minimum operation. After taking the $\min(3, 4, 6) = 3$, we divide the global costs by the number of ranks and by the minimum, 3, to obtain the threshold: 7200 `cells` / 8 `ranks` / 3 `trees per rank` = 300 `cells per tree`. Lastly, `doesLocalTreeViolateThreshold()` yields true as at least one of the three local trees exceeds the threshold, and the strategy initiates rebalancing.

Other than the additional `doesLocalTreeViolateThreshold()` condition, the state transitions are remarkably similar to those of the recursive bi-partition provided in Algorithm 1. Before investigating the state transitions of this strategy, some functionality was abstracted again, see Algorithm 5.

The function `canSplitSOT()` shortens later conditional checks for the split-oversized-tree strategy, as these condition checks are performed before triggering any split to a remote rank. Analog to Algorithm 2, the conditions ensure that, from the perspective of the load-balancer, nothing interferes with instructing a split on the oversized tree. The return statement of `canSplitSOT()` again involves the evaluation of multiple conditions. Only the differences from Algorithm 2 are explained. The third condition checks whether the passed spacetree is sufficiently large in order to perform a split. Lastly, the computed number of ideal splits

Algorithm 4: `updateState` of Split Oversized Tree

```
Input:  state, roundRobinToken
Output: state, roundRobinToken
1  Function updateState(state,roundRobinToken):
2      if state = "SwitchedOff" then
3          and heaviestSpacetree $\neq -1$ // do not update
4      else if areRanksUnemployed() then
5          state $\longleftarrow$ "InterRankDistribution"
6      else if isInterRankBalancingBad() and isMyTurn(roundRobinToken)
7      and doesLocalTreeViolateThreshold() then
8          state $\longleftarrow$ "InterRankBalancing"
9      else if isIntraRankBalancingBad()
10     and doesLocalTreeViolateThreshold() then
11         state $\longleftarrow$ "IntraRankBalancing"
12     else
13         state $\longleftarrow$ "WaitForRoundRobinToken"
14         roundRobinToken $\longleftarrow$ (roundRobinToken $+ 1$) % getNumberOfRanks()
15     return state,roundRobinToken
```

Algorithm 5: Utility function for Split Oversized Tree

```
Input:  spTree, state
Output: boolean
1  Function canSplitSOT(spTree,state):
2      return fitsIntoMemory(state) and not isBlacklisted(spTree)
3             and getCostOfLocalTree(spTree) $>$ getTargetTreeCost()
4             and computeNumberOfSplits(spTree) $> 0$
```

must exceed 0, i.e., splitting the tree is considered useful by the strategy.
If all these conditions hold, a split will be instructed. Next, let us focus on the decision-making of the split-oversized-tree strategy as depicted in Algorithm 6.

Algorithm 6: `updateLoadbalancing` of split-oversized-tree strategy

```
Input:   state, config, costMetric
Result:  triggering a tree-split

1  Function updateLoadbalancing(state,config,costMetric):
2      heaviestSpacetreeTol ⟵ getIdOfHeaviestLocalSpacetreeInTolerance()
3      heaviestSpacetree ⟵ getIdOfHeaviestLocalSpacetree()
4      numLocalUnrefCellsOfHeavSpacetree ⟵ getWeightOfHeavLocSpacetree()
5      switch state do
6          case "InterRankDistribution" do
7              if canSplitSOT(heaviestSpacetreeTol, state)
8              and heaviestSpacetreeTol ≠ −1
9              and getLightestRank() ≠ getMyRank() then
10                 triggerSplit(heaviestSpacetreeTol, getLightestRank(), 1)
11         case "InterRankBalancing" do
12             if canSplitSOT(heaviestSpacetree, state)
13             and heaviestSpacetree ≠ −1 then
14                 triggerSplit(heaviestSpacetree, getLightestRank(),
15                              computeNumberOfSplits(heaviestSpacetree))
16         case "IntraRankBalancing" do
17             for t ∈ getLocalSpacetrees() do
18                 if canSplitSOT(t, state) then
19                     triggerSplit(t, getLightestRank(),
20                                  computeNumberOfSplits(t))
21         otherwise do
               // nothing
```

Note 1: The `triggerSplit(sourceTree, targetRank, numberOfSplits)` function is a wrapper around the `split()` function described in Listing 3.1, but adds strategy-relevant instructions. The function divides the total weight of the source tree by the number of splits plus one to get the number of cells per split. The addition with one accounts for the remaining tree on the current rank. All splits decrease the source tree's weight by the previously calculated number of cells per split.

Note 2: The function `computeNumberOfSplits()` is used for passing the correct argument to `triggerSplit(...)`. It first computes an initial ideal number of splits by dividing the source tree's cost by the ideal tree cost retrieved from `getTargetTreeCost()` and subtracting one to again account for the remaining tree after all splits. This value then is bounded by [1; *remaing slots of the remote spacetree set*][2]. Lastly, the number of splits is decreased until

[2]See Appendix Section A.1, Peano 4 source code is inconsistent with strategy

the first value is obtained for which the cells per split-off tree exceed the configured minimum tree size. This approach is viable as decreasing the number of splits increases the number of cells per tree. The obtained value is finally returned by `computeNumberOfSplits()`.

Like the recursive bi-partition, this strategy is more fit for delayed load distribution. Initial distribution with this strategy may lead to an early uneven load distribution. This behavior comes from the limit that is used to split off trees. Especially during the early grid construction, the local unrefined cells of rank 0 grow exponentially. A more aggressive load-balancing scheme is suitable for this phase of an ExaHyPE 2 application. Once the initial splitting process has progressed and the load is distributed well across all ranks, intra-rank distribution with the split-oversized-tree strategy can yield good results.

3.3.3. Spread-Out

This strategy is kept very simple. Its main idea consists of distributing the total load among all ranks as balanced as possible. For this, a bottom-up-splitting of the trees is used. Inter-rank- or intra-rank balancing is not in the interest of this strategy. After the initial distribution, it does not balance further. The strategy employs each rank with precisely as many trees per rank as deemed suitable in the context of this strategy, w.r.t the thread count. Hence, this strategy is centralized, refer Chapter 1. This unique behavior is realized by querying the current rank ID and switching all worker ranks, i.e., all ranks except for master rank 0, into the stagnation state. The stagnating ranks no longer actively contribute to the load-balancing process but can receive trees from remote ranks, in the case of the spread-out strategy only from rank 0.

The strategy also builds on an initial waiting phase for the best inter-rank distribution. It keeps track of the stable grid iterations, i.e., the number of consecutive recent grid sweeps that did not alter the global number of inner unrefined cells, and only initiates splits when the grid has been stationary for four grid traversals or another more complex condition holds. More on the second condition follows later in Subsection 3.3.5, as the spread-out-once-grid-stagnates strategy lacks this second condition. Unfortunately, the two conditions form a huge downside of the strategy, as a large portion of the grid must be constructed inside a singular spacetree of rank 0 before any load is balanced. This initial grid construction procedure, which suffers from exponential cell growth, should rather rely on multi-core and multi-threaded application principles. Subsection 5.2.3 will delve into the problematic behavior in greater detail.

As this strategy only relies on two internal states, namely inter-rank distribution and stagnation, the previous separately defined functions for updating the state and the load-balancing are both merged into the `updateLoadbalancing(...)` function (cf. Algorithm 7).

Note 1: The if-condition of line 11 in Algorithm 7 evaluates to false for all non-master ranks, comparable to an early return statement appended to line 9. This relates to the aforementioned stagnating behavior of all non-master ranks.

Note 2: For the strategy to work, `getNumberOfTreesPerRank()` is relevant. As the algorithmic implementation of how the number of trees per rank is computed is similar in the previously described `getTargetTreeCost()` function of Subsection 3.3.2, only major differences are mentioned in the following. The function calculates the perfect number of trees per rank. If the grid was stable during the three recent grid sweeps, the code intentionally risks a tree size lower than the user-prescribed minimum tree size: The strategy's primary

Algorithm 7: `updateLoadbalancing` of spread-out strategy

```
Input:   state, config, costMetric, prevNumberOfCells,numberOfStableGridIterations
Result:  state, prevNumberOfCells, numberOfStableGridIterations,
         triggering a tree-split

Function updateLoadbalancing(state,config,costMetric,
                             prevNumberOfCells,numberOfStableGridIterations):
    if prevNumberOfCells = getGlobalNumberOfInnerUnrefinedCells() then
        numberOfStableGridIterations ⟵ numberOfStableGridIterations +1
    else
        numberOfStableGridIterations ⟵ 0
    if not getMyRank().isGlobalMaster()
      and state = "InterRankDistribution" then
        state ⟵ "Stagnation"
    numberOfTreesPerRank ⟵ getNumberOfTreesPerRank()
    if numberOfTreesPerRank > 0 then
        ranks ⟵ getNumberOfRanks()
        cellsPerTree ⟵ round(getGlobalNumberOfInnerUnrefinedCells() /
                                 ranks / numberOfTreesPerRank)
        cellsPerTree ⟵ max(1, cellsPerTree)
        for targetRank ∈ ranks do
            // omitted:  accommodate for rank 0 already hosting one tree
            thisTreesCells = cellsPerTree
            // omitted:  modify thisTreesCells for an imperfectly even
            //           spread, increment by 1 for some trees
            triggerSplit(thisTreesCells,targetRank)
        state ⟵ "Stagnation"
    else
        prevNumberOfCells = getGlobalNumberOfInnerUnrefinedCells()
```

goal is to spread the work as evenly as possible, so this violation is acceptable. For this, if the grid is not predicted to change, the best effort of the spread-out scheme is to distribute the total number of inner unrefined cells perfectly among all trees of all ranks. Second, suppose instead, the grid is still undergoing changes during the last three grid sweeps, but the local number of inner unrefined cells of the master rank root tree is sufficiently large. In that case, all ranks receive their inherent maximum number of trees, bound by user input or thread count, all of equal weight. If neither of the two cases holds, the strategy postpones load-balancing until more cells are refined or more recent grid sweeps are stable.

The `triggerSplit(numberOfCells,targetRank)` in line 17 of Algorithm 7 is a simple wrapper that defaults the source tree to rank 0 and tree ID 0, thus only relying on the cell count and target rank as passed arguments. This load-balancing scheme is predestined to work well on applications that do not use dynamically adaptive mesh refinement but are initially refined during grid construction and then remain unchanged. The strategy is only viable for the initial workload distribution of adaptive mesh refinement simulations, not the full simulation.

3.3.4. Spread-Out-Hierarchically

The strategy follows a similar workflow of the spread-out scheme, refer to Subsection 3.3.3. The main difference is its staged or name-implied hierarchical distribution. First, the focus is put on the spread-out that uses MPI, i.e., the inter-rank spread. A single tree is deployed to each rank. Second, each rank internally balances the received single tree across all its threads. For this, the method only uses two states, inter-rank- and intra-rank-distribution. Algorithm 8 provides the state transitions of spread-out-hierarchically.

Algorithm 8: `updateState` of spread-out-hierarchically strategy

```
Input:   state, config, costMetric
Result:  triggering a tree-split
1  Function updateLoadbalancing(state,config,costMetric):
2      switch state do
3          case "InterRankDistribution" do
4              if getGlobalNumberOfTrees() > 1 or getNumberOfRanks() ≤ 1 then
5                  state ⟵ "IntraRankDistribution"
6          case "IntraRankDistribution" do
7              if getLocalSpacetrees().size() > 1 then
8                  state ⟵ "Stagnation"
9              if getNumberOfThreads ≤ 1 then
10                 state ⟵ "Stagnation"
11         otherwise do
               // nothing
```

Additionally, the strategy uses a set of possible actions dependent on the load-balancer's internal state. The list of actions is: spread equally over all ranks, spread equally over all threads, and none. At the beginning of the update load-balancing function, the next action is determined by a call to `getAction()`. Algorithm 9 describes the decision-making of the spread-out-hierarchically strategy. As the `getAction()` function realizes additional decision-making, its core functionality is described in the following paragraph.

Algorithm 9: `updateLoadbalancing` of spread-out-hierarchically strategy

```
  Input:   state, config
  Result:  triggering a tree-split
1 Function updateLoadbalancing(state,config,costMetric):
2     action ⟵ getAction()
3     switch action do
4         case "SpreadEquallyOverAllRanks" do
5             cellsPerRank ⟵ round(getGlobalNumberOfInnerUnrefinedCells() /
6                                   getNumberOfRanks())
7             cellsPerRank ⟵ max(cellsPerRank,1)
8             for targetRank ∈ getNumberOfRanks()  // Start with targetRank 1
9              do
10                thisRanksCells ⟵ cellsPerRank
                  // omitted:  modify thisRanksCells for an imperfectly even
                  //           spread, increment by 1 for some ranks
11                triggerSplit(thisRanksCells,targetRank)
12        case "SpreadEquallyOverAllThreads" do
13            heaviestSpacetree ⟵ getIdOfHeaviestLocalSpacetree()
14            if heaviestSpacetree ≠ −1
15              and not isBlacklisted(heaviestSpacetree) then
16                numLocalUnrefCellsOfHeavSpacetree ⟵
17                                       getWeightOfHeavLocSpacetree()
18                numberOfSplits ⟵ getNumberOfSplitsOnLocalRank()
19                cellsPerCore ⟵ numLocalUnrefCellsOfHeavSpacetree
20                                / (numberOfSplits +1)
21                cellsPerCore ⟵max(1, cellsPerCore, config.getMinTreeSize())
22                for i ∈ numberOfSplits do
23                    thisCellsPerCore ⟵ cellsPerCore
                      // omitted:  modify thisCellsPerCore for an imperfectly
                      //           even spread, inc.  by 1 for some trees
24                    triggerSplit(thisCellsPerCore,getMyRank())
```

Note 1: `getAction()` of line 2 in Algorithm 9 is not provided as pseudo-code to decrease redundancy to already provided algorithms. To summarize its behavior, it differentiates the internal load-balancing state: In inter-rank distribution, the action *SpreadEquallyOverAll-Ranks* is returned by default. This default action is discarded as soon as some conditions do not hold, e.g., the current action-retrieving rank is not rank 0, or the current trees have been intra-rank balanced in the last three load-balancing updates, or the tree that shall be split is not sufficiently large to deploy at least one minimal spacetree to each rank. If the conditions do not hold, the action *none* is returned. In intra-rank distribution, the program flow is similar: The action *SpreadEquallyOverAllRanks* is returned by default. If a single condition is unmet, this default action is replaced with *none* as a return value. The condition formulates that the local spacetree may not be too lightweight to keep all threads busy and may simultaneously not violate any minimum tree size criteria defined by the user.

Note 2: The `getNumberOfSplitsOnLocalRank()` function is also not elaborated further. The documentation and source code provide implementation details. The returned value is generated by using familiar concepts of other strategies like Subsection 3.3.2 and its internals of the `getTargetTreeCost()` function or Subsection 3.3.3 and its `getNumberOfTreesPer-Rank()` function.

Note 3: The `triggerSplit(numberOfCells,targetRank)` function is a wrapper for the spacetree `split(...)` function like in previous strategies. As the spread-out-hierarchically strategy guarantees that only a single spacetree exists on the current rank when triggering a split, the source tree is implicitly the current rank's sole tree and two passed arguments suffice.

3.3.5. Spread-Out-Once-Grid-Stagnates

This strategy is almost identical to the spread-out strategy. The spread-out-once-grid-stagnates strategy likely served as a predecessor for the improved spread-out strategy described in Subsection 3.3.3. It restricts itself to initiating the perfect tree and rank spread once the grid is stagnating, i.e., the number of stable grid iterations exceeds three. Compared to spread-out, this strategy does not allow splits on a mesh still subject to change. Therefore, it inherits and even worsens the problem of spread-out. The mesh has to be fully built in one spacetree of rank 0 before any rebalancing occurs. Spread-out mitigated this problem by splitting earlier once all ranks could receive their ideal tree counts with a predefined minimum number of local unrefined cells, although the grid is still changing. Subsection 3.3.3 describes calculating the ideal number of trees per rank. The main problem of this strategy stems from the communication during the splitting phase: The MPI that is used to communicate new tree requests to remote ranks has a maximum buffer size. For large meshes built initially, the entire split communication happens between four grid traversals, i.e., all remote ranks receive the requests, the new trees and cells, and all data via MPI buffers simultaneously. This massive communication spike is problematic; look forward to Subsection 5.2.5 for tests of the spread-out-once-grid-stagnates strategy.

3.3.6. Cascade: Spread-Out into Recursive Bi-Partition

Unlike previously introduced strategies, this strategy uses a cascade of load-balancing schemes. The cascade strategy works in the following fashion. First, primary and secondary load-balancing are defined. Initially, the primary strategy spread-out (cf. Subsection 3.3.3) is used for balancing. When any rank's load-balancing stagnates, i.e., the internal load-balancing state is switched to stagnation or switched off, the succeeding recursive bi-partition (cf. Subsection 3.3.1) is activated. Therefore, this cascade spreads the load out in a bottom-up manner and afterward rebalances by recursively bi-partitioning the largest tree on each rank by top-down splitting. This rebalancing with recursive bi-partition after a good initial spread counteracts the above-mentioned problem of large tree size differences when using only the recursive bi-partition strategy. Currently, the implementation does not allow the load-balancing to ascend the cascade again. However, as spread-out only serves for the initial distribution, this is irrelevant. This cascading approach is motivated by one main advantage. A balancing strategy that initially yields great results can be followed by a strategy that works well for later re-balancing.

3.3.7. Cascade: Spread-Out into Split-Oversized-Tree

Like the cascade strategy above, this strategy uses spread-out (cf. Subsection 3.3.3) as the primary load-balancer. After spread-out switches into a stagnating state, the secondary strategy split-oversized-tree (cf. Subsection 3.3.2) is used. Note that, by construction, the use of the primary or secondary load-balancer is a rank-wise decision, so an arbitrary share of ranks can use the secondary strategy while the remaining ranks still use the primary strategy or vice versa. In the case of the spread-out strategy, all ranks initially use the strategy. After the first load-balancing update, only rank 0 is still active and all other ranks have switched to stagnation. Then, the split-oversized-tree strategy is activated to further rebalance any imbalances induced by dynamic workload shifts. This cascade of load-balancing strategies inherits the main advantage of the previously introduced spread-out into recursive bi-partition cascade Subsection 3.3.6. The primary load-balancer is identical to the preceding strategy, and the secondary load-balancer improves load-balancing quality by maintaining the configured maximum tree size.

4. Metrics: Load-Balancing on Dynamically Adaptive Workloads

How can the load-balancing quality of the dynamically adapting space domain be evaluated? For single-core and multi-core multi-threaded applications, load-balancing metrics were developed. The following explains the reasoning behind these metrics and why they help to compare different load-balancing strategies of ExaHyPE 2. Note that these load-balancing metrics, although tailored to ExaHyPE 2, can be transformed to measure other multi-core and multi-threaded applications. Therefore, the following analysis is of broader relevance than only for ExaHyPE 2 applications.

Let $\mathcal{P}$ be the set of MPI program instances and p_i be the i-th program instance in the context of MPI. Furthermore, p equals the number of MPI processes running simultaneously. Additionally, let $\mathcal{T}$ be the set of trees hosted on a previously determined rank and τ_j be its j-th tree. Although described by a mathematical set, a total order is implied for the elements of $\mathcal{P}$ and $\mathcal{T}$.

$$\begin{aligned}
\mathcal{P} &= \{\, p_i \mid 0 \leq i < p, \quad \text{with } i, p \in \mathbb{N}\}, \quad \text{for the number of processes or compute nodes } p \\
\mathcal{T} &= \{\, \tau_i \mid 0 \leq j < \tau, \quad \text{with } j, \tau \in \mathbb{N}\}, \quad \text{for the number } \tau \text{ of trees per node}
\end{aligned}$$

By this definition $|\mathcal{P}| = p$ and $|\mathcal{T}| = \tau$. Furthermore, let $\tau^* \in \mathcal{T}$ be the tree and $p^* \in \mathcal{P}$ the process with the maximum number of local unrefined cells. The problem of measuring the quality of load-balancing for ExaHyPE 2 can be divided into two sub-problems: intra-rank load-balancing and inter-rank load-balancing quality. Solving these problems answers how well the load is balanced within one process, intra-rank, and how well the load is balanced among multiple processes, inter-rank.

4.1. Intra-Rank Metric for Comparing MPI Programs

First, the intra-rank load-balancing quality problem is resolved. Thus, a metric $\mathcal{M}_{p_i}$ was designed for a single MPI compute node i. Although this problem is considerably simpler, the fundamental idea is transferable to the superordinated inter-rank problem. This metric solely relies on the number of local unrefined cells *luc* of a spacetree.

4.1.1. Idea and Realization

Following the keep-it-simple concept, the metric uses the average number of local unrefined cells as a baseline. The reasoning behind using the *luc* count is that this number directly correlates to the total workload a spacetree imposes on the simulation. As the cells are local and unrefined, meaning they have no further refined cells hosted at the same space and thus on a lower spacetree level, they result in a workload per time step for the program.

The MPI intra-rank metric $\mathcal{M}_{p_i}$ should ideally penalize imbalanced and reward balanced load distribution. Therefore, it is generated for a single MPI-program instance p_i by summing over all load-balancing statistic-dump time steps Θ_i concerning the rank i that the process runs on. $\mathcal{T}^*_{t,i}$ denotes the spacetree of time step t and process i that hosts the maximum number of local unrefined cells. $\text{avg}_{x \in \mathcal{T}_{t,i}}(luc(x))$ describes the average local unrefined cells of time step t and rank i, where x is a single spacetree and only used for indicating elements of the set $\mathcal{T}_{t,i}$.

$$\mathcal{M}_{p_i} = \frac{\sum_{t \in \Theta_i} luc(\tau^*_{t,i}) \, / \, \text{avg}_{x \in \mathcal{T}_{t,i}}(luc(x))}{|\Theta_i|} - 1 \tag{4.1}$$

Per timestep $t \in \Theta_i$, the maximum deviation of the average local unrefined cells of rank i is calculated, summed up, and then averaged over the entire time domain of load-balancing statistic dumps.
Thereby, if a compute node only hosts spacetrees with equal numbers of *luc*s, a perfectly distributed workload, the time-step-wise quotient of the maximum and average *luc* repeatedly evaluates to 1, resulting in $(|\Theta_i| \cdot 1)/|\Theta_i| - 1 = 0, \; \forall \; \Theta_i$ as an overall perfect load-balancing score. Otherwise, this metric yields strictly positive values for an imperfect load-balancing scenario. The final goal of all load-balancing schemes is to minimize the metric, $\min_{\mathcal{T}} \mathcal{M}_{p_i}$. The values can then be interpreted as shown in the following example:

Let rank 0, e.g., host four spacetrees. After the ExaHyPE 2-specific application was executed and post-processed, a metric value of $\mathcal{M}_{p_0} = 0.3$ is generated. The value of 0.3 states that, on average, there is at least one of the four intra-rank spacetrees, which has 30% more *luc*s than the average spacetree could have and thereby imposes 30% more computational effort during the same time window.

4.1.2. Justification and Faulty Alternatives

The metric of Equation 4.1 has proven effective as a tool for analysis. Nonetheless, other considerations have been initially made but proven insufficient. For example, another metric followed first had the maximum relative deviation of the average of Equation 4.1 changed to the maximum relative deviation to the second highest *luc* count of the rank. This idea was proven faulty quickly as if there are two spacetrees that host a significantly larger share of total *luc*s than the average, this metric can indicate good load-balancing if and only if the two spacetrees have similar cell counts. Additionally, the interpretability of this metric was bad as the information of the average *luc* count is lost during computation.

For the reason of missing interpretability, an absolute cell metric yielded unusable results, too: If $\mathcal{M}_{p_i}$ of Equation 4.1 yielded the total surplus cell count and had the numerator changed to a difference instead of a quotient, the metric again loses the relation to the average cell counts and thus becomes uninterpretable. E.g., let such a metric $\mathcal{M}_{p_i}$ yield 23 if the metric describes the maximum absolute deviation of the average by subtracting the average and not the maximum relative deviation by dividing as currently used in Equation 4.1. A statement about this value can be made: An average of 23 additional *luc*s are hosted on τ^* for each time step.

However, the question of how well the underlying load-balancing scheme performs remains unanswered; the value of 23 does not indicate the quality of the load-balancing used. If the average *luc* count is 5000, then the used load-balancing strategy yields nearly perfect results; otherwise, for an average of only 5, it yields results far from perfection.

Another initially promising concept was summing up all positive deviations of the average *luc* count and not using only the cell count of τ^* as an indicator. Without respecting the intra-rank tree count, this method would lose the magnitude of how much the average is exceeded from the single most compute-intense and thereby impactful spacetree. Thus, if all local spacetrees are processed in parallel, the second-to-maximum or third-to-maximum spacetree is not performance-critical. In a parallel environment, their increased computation time regarding the average is shadowed by the heaviest spacetree's computation time w.r.t. the *luc* count.

Alternatively, if the metric considers the intra-rank tree count, the metric would distribute the cell count of τ^* overall spacetrees and thereby downplay the gravity of the worse-balanced spacetree on the run-time. The multi-threaded execution for all but one thread, which works on τ^*, may be stalling. Hence, the domain decomposed partial grid sweep performed by the current rank is held back. However, it should be acknowledged that the stall may not be guaranteed:

Investigate a sample scenario with one rank, four spacetrees, and two threads. The *luc* distribution follows 200, 800, 500, 500 for τ_0 to τ_3. If thread one traverses τ_0 followed by τ_1 and thread two traverses τ_2 and τ_3, the uneven load-balancing is accounted for by the threading and does not impose stalling threads. To avoid situations that rely on this probabilistic behavior, the metric of Equation 4.1 would punish such a distribution as if there was stalling introduced by the quadruple cell count of τ_0 over τ_1.

4.2. Inter-Rank Metric for Comparing MPI Programs

After solving the intra-rank metric problem and moving up the MPI+X scheme, the next task is balancing among processes. This problem adds a challenge, as the processes run independently except for synchronization efforts realized with MPI. Thus, the final ExaHyPE 2 log generated line-by-line by different processes interleaves the logs of all processes. Additionally, due to the uncoupled characteristic of processes until the next synchronization, the log lines previously used for generating the load-balancing are dated to different timestamps among the various processes. Nonetheless, synchronization efforts after each grid sweep of Peano 4 guarantee that no rank dumps their statistics of the succeeding sweep before all other ranks have dumped their statistics of the current sweep. In other words, if p_i has dumped its load-balancing statistics of e.g., time step 25, $\Theta_{i,25}$, then all other ranks $\mathcal{P} \setminus p_i$ have already dumped their load-balancing statistics of time step 24, $\Theta_{j,24}$ with $j \neq i \land 0 \leq i < |\mathcal{P}|$. As this holds for arbitrary timestamps during the simulation, the property can be leveraged for an inter-rank metric.

4.2.1. Idea and Realization

This characteristic can be used to generate an inter-rank metric. The sum of all *luc*s over all ranks is used as a baseline. We again use the concept of maximum relative deviation to the average, this time the average *luc*s per rank.

$$\mathcal{M}_{\mathcal{P}} = \frac{\sum_{t\in\Theta} luc(p_t^*)/\text{avg}_{x\in\mathcal{P}_t}(luc(x))}{|\Theta|} - 1 \tag{4.2}$$

For this metric, $\mathcal{M}_{\mathcal{P}} >= 0$ holds, with $\mathcal{M}_{\mathcal{P}} = 0$ being the best possible result, a perfectly even spread among all processes. A value of, e.g., 0.4 can be interpreted as at least one process that hosts the maximum amount of *luc*s, 40% more than the average.

4.2.2. Justification and Faulty Alternatives

This metric $\mathcal{M}_{\mathcal{P}}$ works analogously to the previously developed $\mathcal{M}_{p_i}$. Note, the set of timestamps Θ does not contain single timestamps as elements that hold for all processes, but rather, all elements of the set are p-tuples for the number p of processes. A tuple entry j denotes the time by which the process p_j has dumped its load-balancing statistic. As these time differences are negligible and not of further relevance, a p-tuple can be interpreted as a global timestamp for simplicity.
Per timestamp in Θ, the metric calculates the average local unrefined cells over all processes $\mathcal{P}$ and sums up all relative deviations of this average. The process-internal spread of the *luc*s among possibly multiple spacetrees is neglected, as this can be analyzed with the metric $\mathcal{M}_{p_i}$ of Equation 4.1. For the quality of load-balancing among processes, the total *luc* count per process is relevant. As the metric works almost identical to the $\mathcal{M}_{p_i}$, the faulty alternatives are already mentioned in Subsection 4.1.2.

4.3. Additional Metric for the Quality of Internal Balancing

During the analysis of the metrics mentioned above, one problem arose. Although both serve as a base for comparing different strategies, each one does not reveal more details about the internal balancing than the maximum spacetree-wise or process-wise *luc* count. The bottleneck effect of the maximum candidate grants the before-introduced metrics their relevance for comparison but not for individual evaluation of load-balancings.
This behavior is problematic and can be countered using an additional well-known metric of statistics, the standard deviation, $\boldsymbol{\sigma}$. For a population of size n with mean μ and data points x_i, the standard deviation $\boldsymbol{\sigma}$ of the population is defined as:

$$\boldsymbol{\sigma} := \sqrt{n^{-1} \cdot \sum_{i=0}^{n-1} (x_i - \mu)^2}$$

4.3.1. Intra-Rank Metric: Standard Deviation of the Average

By these means, we can leverage this metric on a local (intra-process) level to evaluate the spread among spacetrees of a rank. Following the previous naming convention, the metric $\boldsymbol{\sigma}_{p_i}$ measures the intra-rank spread with $\boldsymbol{\sigma}_{p_i} = 0$ describing a perfect intra-rank load-balancing. Over all time steps Θ_i of rank i, $\boldsymbol{\sigma}_{p_i}$ is defined as the average standard deviation from the timestamp-wise average local unrefined cells of rank i:

$$\boldsymbol{\sigma}_{p_i} := \sum_{t\in\Theta_i} \sqrt{|\mathcal{T}_{i,t}|^{-1} \cdot \sum_{x\in\mathcal{T}_{i,t}} (luc(x) - \text{avg}_{y\in\mathcal{T}_{i,t}} luc(y))^2} \quad / \quad |\Theta_i| \tag{4.3}$$

This metric grants insight into how well a particular rank i internally balances its spacetrees over the entire simulation time. An example of how an analogous metric works can be examined in the following Subsection 4.3.2 and is not repeated.

4.3.2. Inter-Rank Metric: Standard Deviation of the Average

Like $\boldsymbol{\sigma}_{p_i}$, another metric over all ranks can be formulated. The metric $\boldsymbol{\sigma}_p$ measures the inter-process spread with $\boldsymbol{\sigma}_p = 0$ describing a perfect inter-rank load-balancing. Viewed over all time steps Θ, $\boldsymbol{\sigma}_p$ is defined as the average standard deviation from the timestamp-wise average local unrefined cells:

$$\boldsymbol{\sigma}_p := \sum_{t\in\Theta} \sqrt{|\mathcal{P}_t|^{-1} \cdot \sum\nolimits_{x\in\mathcal{P}_t} (luc(x) - \mathrm{avg}_{y\in\mathcal{P}_t} luc(y))^2} \quad / \quad |\Theta| \tag{4.4}$$

This metric adds a decision criterion about the internal quality of the load-balancing, which is not fully focused on comparability. The impact of this combined analysis can be examined on an inter-rank analysis of three exemplary time-slices with ten processes each. In the cases below, $|\Theta| = 1$, i.e., there is only one timestamp, so the summation over all timestamps and division by $|\Theta|$ are neglected. For the following list, the notation $p_i = c$ assigns process i a number c of local unrefined cells, ignoring the internal multi-thread spacetree spread.

1. *luc* split: $p_0 = 2500, \quad p_1, p_2, \ldots, p_8 = 1000, \quad p_9 = 0$
   - avg = 1050
   - Max. rel. deviation = $\mathcal{M}_\mathcal{P} = 2500/1050 - 1 \approx 1.38$
   - $\boldsymbol{\sigma}_p = \sqrt{10^{-1} \cdot ((2500 - 1050)^2 + 8 \cdot (1000 - 1050)^2 + (0 - 1050)^2)} \approx 567.89$
2. *luc* split: $p_0 = 2500, \quad p_1, p_2, \ldots, p_8 = 900, \quad p_9 = 800$
   - avg = 1050
   - Max. rel. deviation = $\mathcal{M}_\mathcal{P} = 2500/1050 - 1 \approx 1.38$
   - $\boldsymbol{\sigma}_p = \sqrt{10^{-1} \cdot ((2500 - 1050)^2 + 8 \cdot (900 - 1050)^2 + (800 - 1050)^2)} \approx 484.25$
3. *luc* split: $p_0 = 1500, \quad p_1, p_2, \ldots, p_5 = 1200, \quad p_6, p_7, p_8 = 1000, \quad p_9 = 0$
   - avg = 1050
   - Max. rel. deviation = $\mathcal{M}_\mathcal{P} = 1500/1050 \approx 0.43$
   - $\boldsymbol{\sigma}_p = \sqrt{10^{-1} \cdot ((1500 - 1050)^2 + \ldots + (0 - 1050)^2)} \approx 377.49$

Note, the max. rel. deviation of the cases above describes the relative deviation of the maximum *luc* count from the average and does not necessarily describe the maximum relative deviation from the average. In cases 1 and 2, these yield identical results, but for case 3, the maximum relative deviation from the average would be the deviation of p_9, which has an absolute deviation of -1050. This behavior is intentional, as in a load-balancing context, only the wrongly balanced compute-heavy processes impose a performance bottleneck.
When viewing cases 1 and 2 from above, the preceding metric $\mathcal{M}_\mathcal{P}$ collapses to the maximum relative deviation as only one timestamp is analyzed. The metric yields identical results for both cases. Therefore, if the processes run perfectly parallel without scheduling or memory

dependencies, they are bottlenecked by $p_0 = p^*$ and finish the time step simultaneously. From a user perspective, the *luc* spread among processes has no further impact. However, this spread is substantial for even load distribution, equal hardware usage, and overall evaluation of superior load-balancing schemes. The lower standard deviation $\boldsymbol{\sigma}_p = 484.25$ indicates a better spread and thus a more dense distribution of the recorded *luc* spread around the average. Case 3 is, by construction, the best, as the lower maximum relative deviation leads to a significant reduction of sum value, although mitigated by the subsequent application of the square root. The maximum relative deviation of case 3 already implies a better-performing *luc* split. The bottlenecking $luc(p_0)$ are closer to the average *luc* count, and the standard deviation from the average is significantly smaller than in cases 1 and 2.

5. Comparison of Load-Balancing Strategies

5.1. Description of Test Cases

In the following, the hardware specifications used for the testing are listed. As this is not a performance test but a load-balancing test, the underlying hardware has no effect on the result; further hardware information is spared.

- CPU: AMD EPYC 7402 @ 2.80GHz, 24 cores, 128 MiB L3 Cache.
- RAM: 256 GiB
- Storage: 512GiB NVMe PCIe Gen4 SSD

The example 3D scenario named LOH1 was chosen for the load-balancing performance audit. LOH1 is commonly used to verify the accuracy of seismic wave propagation codes. It models a sediment layer over a base material. The outgoing waves from a point source in the base material propagate through the heterogeneous material. We use this real-world example to test the load-balancing behavior for future applications. The chosen uniform cartesian grid for the LOH1 benchmark consists of 19683 cells with a discretization order of 7. The 12 variables hosted per cell result from three time-independent material parameters and nine additional unknowns dependent on time for simulating wave movement. The described problem has a total of 121 million degrees of freedom. The most relevant information for evaluating load-balancing schemes is the total number of 19683 unrefined cells, which should be ideally balanced among all ranks and their threads.

During the testing phase, many load-balancing strategies resulted in suboptimal behavior, even with AMR disabled. In the scope of this thesis, runs with AMR would have reduced the interpretability of the resulting load distributions by adding high-complexity refining and coarsening operations. As the succeeding sections will show, the runs without AMR also give insights into strategy behavior. However, the metrics of Chapter 4 were designed to evaluate AMR applications' load-balancing statistics. Testing non-AMR applications does not present the metrics' full potential. The decision about disabling AMR was made to facilitate linking the following load-balancing *luc* distributions to the algorithmic behavior of the strategies. Testing and evaluating applications with AMR enabled are the subject of future work. The following three test cases were run with different MPI rank and OMP thread numbers but without adaptive mesh refinement.

Case 1: Run a **single program** instance without MPI with **24 OpenMP threads**. This will serve for the intra-rank load-balancing evaluation.

Case 2: Run **2 program** instances (ranks) that use MPI with **12 OpenMP threads** each. The reasoning behind this is to test simple inter-rank balancing and maintain the previously tested intra-rank load-balancing quality.

Case 3: Run **12 program** instances (ranks) that use MPI with **2 OpenMP threads** each. Last, the extreme case of balancing the load between ranks with few threads will grant insights about the quality of inter-rank load-balancing.

After clarifying the test case parameters, all strategies are evaluated using the metrics derived in Chapter 4. These post-processing metrics are applied to the log files after a significant run-time. Many simulations were interrupted before finishing, as the final result of the simulation is not relevant in the load-balancing context. The reason for this early interruption is that many load-balancers are switched off after a specific time, so the balance stays constant afterward. If they are not switched off, it is noted in the corresponding subsection. After the interruption, the simulation was kept running manifold the previous time from start to switch-off. By this approach, all further noted metrics have converged to a value. The first test case may be analyzed to a larger extent than the second and third. This is based on easier and more in-depth explanations for a non-MPI run.

5.2. Results of Test Cases

5.2.1. Recursive Bi-Partition

The first test case without MPI resulted in bad splitting behavior. Although 24 threads were provided, the load-balancer only created four trees. One empty fifth tree was also created for nine load-balancing dumps but removed again. The initial behavior seems correct; examine the first logged local unrefined cell counts:

$\|\mathcal{T}\| = \tau$	$\tau_0 \in \mathcal{T}$	$\tau_1 \in \mathcal{T}$	Total *luc*
1	27		27
2	27	0	27
2	27	13	40
2	378	351	729

Table 5.1.: No MPI, initial load-balancing statistic dumps.

An entry of 27 for τ_0 indicates that the tree with ID 0 has 27 local unrefined cells. Recall the recursive bi-partition's main idea: Split trees in half repeatedly.

After τ_1 was created in row 2, it received 13 cells from τ_0 in row 3. From row 3 to row 4, two changes happen simultaneously to τ_0: The tree transfers ownership of the 13 cells to τ_1 and afterward refines its remaining 14 cells to $14 * 3^3 = 378$ cells. The 3^3 stem from the Peano curve splitting a single cell into three cells along each dimensional axis. We are considering a three-dimensional problem, so a single cell is refined into 27 finer cells. In the same grid sweep, the 13 new cells of τ_1 are refined to $13 * 3^3 = 351$ cells. The algorithmic behavior seems promising. However, after 30 load-balancing dumps and 19-second run-time, the load-balancing switches off, and the final configuration is depicted in Table 5.2.

The simulation would continue to run with four trees, not exploiting all 24 but only four threads of compute resources and hosting very imbalanced tree weights. Nevertheless, it is worth recognizing that the progression of τ_0's to τ_3's local unrefined cells indicates a recursively bi-partitioned splitting behavior with deviation due to being unable to split 27

τ	τ_0	τ_1	τ_2	τ_3	Total *luc*
4	10206	4752	3299	1426	19683

Table 5.2.: No MPI, final load-balancing statistics dump after switch-off.

cells perfectly in half. The entire behavior must not be unintended, as trees can reach a composition where splitting off further is impossible. Objectively, the load is not balanced well. This can easily be verified with the intra-rank metrics of Chapter 4:

$$\mathcal{M}_{p_0} \approx 1.074 \qquad \boldsymbol{\sigma}_{p_0} \approx 3267 \tag{5.1}$$

The metric intra-rank $\mathcal{M}_{p_0}$ with a value of more than one means that, on average, the heaviest spacetree per timestep hosts 100% more workload than in a perfectly balanced intra-rank situation. Note that the metric, as currently implemented, does not take non-existent trees of a rank into account, even if the rank provides more threads for computation. This creates a more optimistic metric. If the already optimistic metric yields unsatisfying results, an improved metric that takes the empty threads into account will result in more severe results. Nonetheless, for the four out of 24 threads used, we can see that τ_0 is the maximum spacetree. It hosts 10206 cells instead of the perfect average for four threads, $19683/4 \approx 4921$ cells. If we verify the metric by calculating the surplus of the maximum over the average in percent, we receive $10206/4921 - 1 \approx 1.074$. We can only verify the metric as it has already converged over a long run-time that allows neglecting the initial spreading behavior of the strategy.

Advancing to the second test case, the logged load-balancing statistic dumps are alarming. The initial splitting seems correct across the ranks but later splits exhibit unfavorable *luc* counts. After 11 minutes of run-time, the recursive bi-partition strategy has been stuck in the inter-rank distribution state for the entire time. Then, the load-balancing switches off. For this to happen, the internal bookkeeping must repeatedly evaluate `areRanksUnemployed()` of Algorithm 1 as `true`. The results let us conclude that the algorithm does not behave in the intended way. The individual spacetree sets of both ranks grow continuously. This stems from the load-balancing scheme using the `getLightestRank()` method to determine the target rank for splits during inter-rank distribution. The expression can be evaluated to the local rank ID and trigger intra-rank splits, although the state of the load-balancer is in inter-rank distribution. The load-balancing scheme generally tries to split off cells aggressively over and over (cf. Table 5.3).

Rank	τ	τ_0	τ_1	τ_2	τ_{3-88}	τ_{89}	τ_{90}	Total *luc*
0	91	1944	1905	1905	...	22	22	9864
Rank	τ	τ_0	τ_1	τ_2	τ_{3-126}	τ_{127}	τ_{128}	Total *luc*
1	129	2676	2664	1282	...	20	20	9819

Table 5.3.: 2 MPI ranks, 12 threads each, final load-balancing statistics dump after switch-off.

As visible in Table 5.3, the inter-rank workload is split almost perfectly. This is based on the previously mentioned constant internal state of inter-rank distribution. This allows the load-balancer to balance perfectly between the ranks, with more than 100 splits that split off trees of weight less than 45 local unrefined cells. Then, after almost 11 minutes of run-time, the load-balancing is switched off externally. It is not turned on again, and the spacetree decomposition remains for the rest of the simulation. This intra-rank decomposition is strongly uneven. The load-balancer, which remained in the intra-rank balancing state, indicates unintended behavior. The number of splits required much run-time, and we can evaluate the quality of load-balancing with the previously derived metrics again. First, Equation 5.2 depicts the inter-rank metrics.

$$\mathcal{M}_{\mathcal{P}} \approx 0.006 \qquad \boldsymbol{\sigma}_p \approx 36.95 \tag{5.2}$$

$\mathcal{M}_{\mathcal{P}}$ showcases that the inter-rank spread is almost perfect, as values closer to 0 represent a better spread. $\boldsymbol{\sigma}_p$ is comparably small, corresponding to the almost non-existent deviation of the total local unrefined cell count between the two cells (cf. Table 5.3). However, when examining the intra-rank spread, Equation 5.3 displays the bad spread inside the ranks.

$$\begin{aligned} \mathcal{M}_{p_0} &\approx 14.39 \qquad \boldsymbol{\sigma}_{p_0} \approx 446.4 \\ \mathcal{M}_{p_1} &\approx 29.66 \qquad \boldsymbol{\sigma}_{p_1} \approx 461.9 \end{aligned} \tag{5.3}$$

An intra-rank spread $\mathcal{M}_{p_i}$ evaluating to 14.39 or 29.66 is severely deficient. This indicates that on rank 1, a single tree hosts almost thirty times the perfectly averaged workload inside rank 1. An analog conclusion can be drawn for the $\mathcal{M}_{p_0}$ of approximately 14, where the heaviest tree has a surplus workload of 14 times of the perfect average that a load-balancer strives for. The metrics support the thesis of unsatisfactory spreads.

The third test case reveals further details about inter-rank splitting behavior. Promising inter-rank behavior was already inferable from test case 2, but the new test case indicates contradictory behavior to the preceding tests. In the third test case, the load-balancing is again switched off after a short run-time of 48 seconds. The final load balance can be viewed in Table 5.4.

After test case 3, the load-balancing results seem more practical. The split is still far from perfect, and rank 11 is unused, but the heaviest rank, rank 3, is responsible for $4739/19683 \approx 24\%$ of the total workload. A perfectly even inter-rank distribution distribution would yield a $100\%/12 \approx 8\%$ workload. An almost identical performance to test case 3 could be achieved by quartering the workload perfectly with one rank and four threads or any other composition of ranks and threads whose product yields 4. Applying the developed inter-rank metrics on the logs of the third test case yields the values displayed in Equation 5.4.

$$\mathcal{M}_{\mathcal{P}} \approx 1.934 \qquad \boldsymbol{\sigma}_p \approx 1641.80 \tag{5.4}$$

The $\mathcal{M}_{\mathcal{P}}$ of 1.9 is not as bad as in test case 2, but it is still poor. A value of $\mathcal{M}_{\mathcal{P}} \in [0; 1]$ should be a minimum goal for any inter-rank balancing. Otherwise, a single rank has at least double the workload of the perfectly averaged balance; in our example test case nearly triple the workload. The inter-rank standard deviation $\boldsymbol{\sigma}$ of 1642 also implies the uneven spread among ranks that can be examined in the *Total luc* column of Table 5.4. Additionally, the 12 intra-rank metrics computed are supporting all previous conclusions with the worst

Data / Rank ID	τ	τ_0	τ_1	τ_2	τ_3	Total *luc*
0	1	229				229
1	1	309				309
2	2	95	559			654
3	3	94	2094	2551		4739
4	3	1275	1030	1173		3478
5	3	871	1282	1275		3428
6	4	776	1172	1149	638	3735
7	4	508	253	405	439	1605
8	3	525	79	206		810
9	3	50	304	266		620
10	1	76				76
11	0					0

Table 5.4.: 12 MPI ranks, 2 threads each, final load-balancing statistics dump after switch-off.

intra-rank metrics of rank 3 yielding $\mathcal{M}_{p_3} = 0.611$, and $\boldsymbol{\sigma}_{p_3} = 1063.12$. Rank 3's heaviest tree induces a 60% additional workload on the rank on top of the average load, and its intra-rank deviation is huge.

In conclusion, the recursive bi-partition seems not to be feasible as a standalone load-balancer. It is important to note that this was never the goal of the strategy, as it was designed for later rebalancing, not initially spreading work. Its capability of distributing the work before the rebalancing phase begins is an extension based on the targeted balancing phase.

5.2.2. Split-Oversized-Tree

The first test case without MPI but with 24 threads resulted in no load-balancing. The load was not balanced at all. The entire grid of 19683 cells was built in tree 0 of the single process that runs without MPI. The repeatedly logged threads available to the program at any time were 24, matching the configuration of Section 5.1. The load-balancing is also switched off after 16 seconds of run-time, resulting in the worst-case load-balancing with $|\mathcal{T}| = \tau = 1$ spacetree and $luc(\tau_0) = 19683$ for the entire simulation time. The results are not further analyzed with the custom-designed metrics.

The second test case with 2 MPI ranks and 12 threads each indicated bad thread exhaustion. The inter-rank splitting phase yielded good results, with the master tree of 27 cells employing the second rank with 13 cells. The implemented strategy passes the number of splits to a certain rank into the `triggerSplit(...)` method, refer to Algorithm 6, and then `triggerSplit(...)` calculates the ideal number of cells to split off the tree. In the inter-rank distribution phase, the number of splits to a certain rank defaults to 1, so the tree is halved. This is similar behavior to the spread-out-hierarchically inter-rank distribution phase depicted in Algorithm 9. In the test case, the load-balancing switched off after 19 seconds of run-time. The final spread is displayed in Table 5.5.

	τ	τ_0	τ_1	Total *luc*
Rank 0	2	5103	4752	9855
Rank 1	2	4725	5103	9828

Table 5.5.: 2 MPI ranks, 12 threads each, final load-balancing statistics dump after switch-off.

The strategy again uses the accessible threads poorly. Each rank could host 12 spacetree and sweep through them in a multi-threaded fashion. Instead, only two trees are hosted individually, and ten free threads are wasted per rank. Nonetheless, the split-oversized-tree strategy yields good inter-rank balance. The metrics of Equation 5.5 show the quality of the spread.

$$\mathcal{M}_{\mathcal{P}} \approx 0.008 \qquad \boldsymbol{\sigma}_p \approx 35.66 \tag{5.5}$$

The inter-rank metrics are easy to interpret and display that the ranks are split almost perfectly. The generated intra-rank metrics promise deceivingly great intra-rank spreads (cf. Equation 5.6).

$$\begin{aligned} \mathcal{M}_{p_0} &\approx 0.035 \qquad & \boldsymbol{\sigma}_{p_0} &\approx 172.96 \\ \mathcal{M}_{p_1} &\approx 0.040 \qquad & \boldsymbol{\sigma}_{p_1} &\approx 198.22 \end{aligned} \tag{5.6}$$

This deception is based on the fact that the metric currently does not take the unused threads into account. If, for example, the surveyed simulation had only two ranks with two threads each available, the metrics would indicate very good balancing and would not deceive but state the truth. Altering the metrics to avoid this behavior is a future goal.

Lastly, in the third test case of the split-oversized-tree strategy, the strategy encountered a problem. The previously perfectly run simulation that was used for all tests terminated after 13 seconds due to a floating point exception. For future analysis, the error produced should be reproduced and looked into. The identical preceding setup for all tests without mathematical errors raised leads to the suspicion that the load-balancer may perform the exception-raising calculation. Therefore, further load-balancing spread evaluations are omitted as the run-time is too short to draw conclusions. Additionally, the load-balancing has not switched off yet, so conclusions can yield wrong positive or negative impressions. The raised exception should be analyzed and circumvented before using this strategy on loads.

5.2.3. Spread-Out

The spread-out strategy yielded great results in the first test case using only 24 threads and a single process. As the initial space domain is first subdivided into $3^3 = 27$ cells, these are immediately spread across all 24 threads. This results in τ_0, τ_{4-23} receiving one and $\tau_{1,2,3}$ receiving two cells each. The trees with two cells each account for the division of 27 by 24 with the remainder of 3. The remainder is spread among the first three trees that the load-balancer spreads to. This algorithmic behavior that respects the remainder was pointed to in a comment in lines 16 and 17 of Algorithm 7: `// omitted: modify thisTreeCells for an imperfectly even spread, [...]`.

Afterward, the strategy does not act any further as the cells are perfectly spread. They then refine to the maximum refinement level and yield the following intra-rank distribution (cf. Table 5.6).

τ	$\tau_{0,4,5,\ldots,23}$	$\tau_{1,2,3}$	Total *luc*
24	729	1458	19683

Table 5.6.: No MPI, final load-balancing statistics dump after switch-off.

Note, the entries in this notation denote that each of the trees listed in the line above has an identical amount of cells. The entries do not denote the summed *luc*s of all tree IDs listed above. The reason why spread-out already decided to split when only having 27 cells in the source tree is explained in Subsection 3.3.5, as it is based on the single condition that spread-out uses which spread-out-once-grid-stagnates does not.

To recapitulate the spread-out strategy, it distributes load even if the grid has not been stationary for more than three grid sweeps. This only happens when the source spacetree can supply the required number of trees with a minimum tree size. As the minimum tree size was not explicitly set in the initial configuration of the tests, it defaulted to 1 during internal computations of the strategy. The 27 cells supplied are more than the 24 needed to distribute the work successfully.

This behavior may not be ideal in this case, but providing a minimum tree size of 2 would have already led to a postponement of the distribution until the next, in our test case second to last, refinement took place so the distribution results would improve due to a finer spacetree decomposition. The perfect balance after waiting for the grid to be fully constructed can also be examined in Subsection 5.2.5, where the condition leading to the earlier distribution of spread-out is not part of the spread-out-once-grid-stagnates strategy. Applying the metrics of Chapter 4 yields the metrics that can be inspected in Equation 5.7.

$$\mathcal{M}_{p_0} \approx 0.797 \qquad \boldsymbol{\sigma}_{p_0} \approx 240.29 \tag{5.7}$$

The inter-rank metric $\mathcal{M}_{p_0}$ of the single program instance that was run showcases that each of the three trees $\tau_{1,2,3}$ induces an additional 80% workload in between load-balancing updates. This stems from the already mentioned remainder of the division and can only circumvented by increasing the minimum tree size. The standard deviation metric implies a rather dense distribution around the average.

The results of the first test case are reproduced in the second test case with 2 MPI ranks and 12 threads. The previous 24 spacetrees of one rank are now spread onto the two available ranks. The load-balancing is switched off after 4 seconds of run-time, and the exact cell distribution can be examined in Table 5.7.

The distribution, respecting algorithmic decision-making, seems perfect once again. The τ_{1-3} have increased cell counts for the same reminder wrap-around behavior pointed out before for test-case one. The metrics can be inspected in Equation 5.8.

$$\mathcal{M}_{\mathcal{P}} \approx 0.113 \qquad \boldsymbol{\sigma}_{p} \approx 1088.69 \tag{5.8}$$

Rank ID	τ	τ_0	τ_1	τ_2	τ_3	τ_{4-11}	Total *luc*
0	12	729	1458	1458	1458	729	10935
1	12	729	729	729	729	729	8748

Table 5.7.: 2 MPI ranks, 12 threads each, final load-balancing statistics dump after switch-off.

The metrics yield good results, with only an 11% average relative deviation from the average cell count. The standard deviation is comparably large but does not add significant insight in the case of only two ranks. The spread across only two ranks can be judged with $\mathcal{M}_{\mathcal{P}}$ as well.

When examining the results of the final test case, the execution does not run smoothly. In the first second of run-time, the entire logging halts. Then, after 1 hour of not logging, the program terminates with an MPI error message because rank 8 invoked an MPI abort after waiting for an MPI `reduce()` call to return. The final retrievable load-balancing distribution is summarized in Table 5.8.

Data \ Rank ID	0	1	2,. . . ,11
τ	2	2	2
τ_0	27	54	27
τ_1	54	54	27
Total *luc*	81	108	54

Table 5.8.: 12 MPI ranks, 2 threads each, final load-balancing statistics dump after switch-off.

This distribution was reached within the first second of the simulation and does not represent the inter-rank quality achieved by this strategy. In the few logs generated, the load-balancing has not been switched off yet, and the cells must still be refined further to meet the cell count of the fully refined grid. Revision of this test case and looking into the unwanted MPI timeout is a future goal, as this strategy yielded perfect initial distribution if it is tuned with the minimum tree size.

5.2.4. Spread-Out-Hierarchically

In this subsection, the next strategy is evaluated. To recapitulate, the unique behavior of this strategy was to let the master rank only deploy one tree to each rank and successively let the ranks themselves split the tree into the desired tree count per rank. In the first test case without MPI and with 24 threads, this behavior should equal the spread-out behavior of Subsection 5.2.3. After confirming the splits, the sequence of splitting and refinements yielded an identical distribution to the spread-out strategy. The metrics and tree splits are not repeated; review Subsection 5.2.3 for the first test case results.

The second test case used 2 ranks and 12 threads per rank. First, the master-rank tree should be split in half, and the half should then be deployed to the second rank. After this, both ranks should simultaneously refine their cells and split them to each thread. When reviewing the logged splits, we can examine the described behavior. First, τ_0 of rank 0 has 27 cells. Then, it splits off 14 cells to τ_0 of rank 1. Afterward, both trees fully refine the cells of the single spacetree they host and finally split them across their threads. The final spread after turning off the load-balancing can be examined in Table 5.9.

Rank	τ	$\tau_{0,10,11}$	τ_{1-9}	Total *luc*
0	12	850	851	9477

Rank	τ	$\tau_{0,7-11}$	τ_{1-6}	Total *luc*
1	12	789	790	10206

Table 5.9.: 2 MPI ranks, 12 threads each, final load-balancing statistics dump after switch-off.

The inter-rank distribution is good, but the intra-rank distribution is perfect. The inter-rank distribution can be measured with $\mathcal{M}_{\mathcal{P}}$ of Equation 5.9.

$$\mathcal{M}_{\mathcal{P}} \approx 0.038 \qquad \boldsymbol{\sigma}_p \approx 364.01 \tag{5.9}$$

$\mathcal{M}_{\mathcal{P}}$ of almost 0.04 showcases that ranks 0 and 1 have total cell counts that differ from the average by only 4% of the average. This is an indication of a very good split. The $\boldsymbol{\sigma}_p$ is adequately small as deviating 4% from the average equals an absolute cell count of 394 cells. The $\boldsymbol{\sigma}_p$ value of less than 394 matches the inter-rank metric. Additionally, Equation 5.10 showcases the perfect intra-rank spreads of both ranks individually.

$$\begin{aligned} \mathcal{M}_{p_0} &\approx 0.005 \qquad & \boldsymbol{\sigma}_{p_0} &\approx 1.95 \\ \mathcal{M}_{p_1} &\approx 0.006 \qquad & \boldsymbol{\sigma}_{p_1} &\approx 2.14 \end{aligned} \tag{5.10}$$

Nothing can be added to these scores, spread-out-hierarchically performed perfectly in the intra-rank distribution.

The third test case with 12 ranks and 2 threads per rank is a prime example of how well a load-balancing strategy can work in the distribution phase. After 6 seconds run-time, the load-balancing was switched off, and the last statistics dumps resulted in the distribution that can be examined in Table 5.10

	τ	τ_0	τ_1	Total *luc*
Rank 0, 4-11	2	729	729	1458
Rank 1, 2, 3	2	1093	1094	2187

Table 5.10.: 12 MPI ranks, 2 threads each, final load-balancing statistics dump after switch-off

With this strategy, most ranks received one tree with two cells of the master ranks' 27 cells first. As 27 is not perfectly divisible by 12, the rank count, the remainder of three is again distributed to three of the 2-cell trees. These are the initial trees of rank 1-3. Then, the trees are refined and split perfectly across the threads. The metric score only suffers from the three initial cells that had to be further distributed. The intra-rank metrics for this strategy can be examined in Equation 5.11.

$$\mathcal{M}_{\mathcal{P}} \approx 0.354 \qquad \boldsymbol{\sigma}_p \approx 315.01 \tag{5.11}$$

Even though the strategy acted perfectly, a workload increased by 35% over the perfect inter-rank average is induced by one of the trees $\tau_{0,10,11}$. The strategy could have only acted better if a minimum tree size of 3 was set so that the initial 27 cells would not have sufficed to deploy 12 trees to ranks with a tree size greater than 3. Intra-rank metrics are omitted as they are extremely close to 0, and the intra-rank distribution is perfect.

In conclusion, the spread-out-hierarchically strategy has the potential to perform very well if the minimum tree size is tuned. Even without tuning, the results were promising for future experiments and development that can build on this strategy.

5.2.5. Spread-Out-Once-Grid-Stagnates

Last of all standalone load-balancers, the spread-out-once-grid-stagnates strategy is tested. To recapitulate, the strategy spreads if and only if the grid was stationary for more than three load-balancing updates and should theoretically achieve overall better results than the spread-out strategy tested in Subsection 5.2.3. On the first test case without MPI but with 24 threads, the strategy yielded a distribution close to perfection after 27 seconds and switching the load-balancing off (cf. Table 5.11).

τ	τ_0	τ_{1-23}	Total *luc*
24	823	820	19683

Table 5.11.: No MPI, final load-balancing statistics dump after switch-off.

The metrics for this case are spared as they will again show values close to 0 and do not reveal any more insights.
The second test case that uses 2 MPI ranks and 12 threads per rank again yielded great results. The final distribution after 28 seconds and switching off the load-balancing is displayed in Table 5.12.

	τ	τ_0	τ_{1-11}	Total *luc*
Rank 0	12	823	820	9843
Rank 1	12	820	820	9840

Table 5.12.: No MPI, final load-balancing statistics dump after switch-off.

The results are perfect. These perfect results are only possible as the entire grid of 19683 cells is built in the initial spacetree of master-rank 0. Afterward, it is perfectly decomposed in one giant effort. A problem that is later addressed further is the scalability of this approach. The metrics are spared for the reason that this distribution is perfect.

The third test case with 12 ranks and 2 threads per rank yielded perfect results again. The grid was fully built on the single tree of the master rank and then split equally across all ranks. The resulting split after 26 seconds of run-time and switching off the load-balancer are displayed in Table 5.13.

	τ	τ_0	τ_1	Total *luc*
Rank 0	12	823	820	1643
Rank 1-11	12	820	820	1640

Table 5.13.: No MPI, final load-balancing statistics dump after switch-off.

The metrics are again omitted as the spread is perfect. In conclusion, the strategy achieved the highest scores across all tests. However, this comes with a large drawback. The construction of the entire grid in a singular spacetree of rank 0 does not use all available compute resources. It is constructed in a serial fashion. This approach is not viable when building grids for exascale simulations. The grid construction for significantly finer resolved grids takes too much time as the total grid cell count grows exponentially following the formula $(3^3)^l = 27^l$ for each further refinement level l. Therefore, serial grid construction is impossible. During extensive testing, this scalability problem arose quickly. Even if this strategy yields perfect results for comparably small grids, it lacks parallelism.

5.2.6. Cascade: Spread-Out into Recursive Bi-Partition

The first cascading strategy uses a combination of spread-out and recursive bi-partition. In the first test case without MPI, the initial spread-out strategy balanced the work identically to Table 5.6. The switch from spread-out to recursive bi-partition is also logged, but the three heavy-weight trees are not split further by the recursive bi-partition strategy. Then, the load-balancing is switched off. Thus, in the tested case, the results do not differ from the first test case of the spread-out strategy.

In the second and third test cases, unintended behavior prevents assessing load-balancing results. The logging again stops within the first second of run-time, and in both test cases, an MPI abort error intentionally raised by a rank causes the simulation to halt after 1 hour of tracked deadlocking. This repeated MPI deadlocking behavior suggests a flaw in the load-balancer's explicit inter-rank communication where, e.g., global cell counts are exchanged.

The spread-out and recursive bi-partition interaction seems to let the error occur more frequently. Nonetheless, the MPI abort was also caused by the standalone spread-out test with 12 ranks and 2 threads (cf. Subsection 5.2.3). In conclusion, the cascading of spread-out and recursive bi-partition currently yields unsatisfying, unavailable load-balancing results.

5.2.7. Cascade: Spread-Out into Split-Oversized-Tree

The first test case for this strategy again resulted in promising results, although creating more trees than threads available. Initially, spread-out spreads the work identical to the distribution displayed in Table 5.6. Three trees host twice as many cells as the other trees. Then, after spread-out switches into stagnation, the successor split-oversized-tree is activated. It tries to rebalance early, during the phase where the initial split-off trees of spread-out have not been refined yet. Thus, the three oversized trees with two *luc*s in comparison to the others with one *luc* are detected as oversized trees. The strategy creates three additional trees on the rank, $\tau_{24,25,26}$. Initially, they could not be filled with cells and were removed from the spacetree set after six load-balancing iterations. Then, after the 24 spread-out trees have refined to the maximum, the three oversized trees that host 1458 cells each are split in half and deployed to a new tree. This time, the splitting process succeeds, and the strategy achieves a split where each of the 27 spacetrees hosts 729 cells. It's a perfect spread. However, creating a slight excess of trees on a local node exceeds the true parallel capabilities of the node and does not improve performance. The excess requires the three extra trees to wait until the traversals of three other trees have finished. The load-balancing is switched off afterward, and the 27 trees persist for the rest of the simulation. Metrics are omitted as they would indicate perfect load-balancing, neglecting the overloaded threading parallel capabilities. As mentioned, reworking the metrics' behavior to consider the thread count is a future goal but is out of the scope of this thesis.

The second test case with 2 ranks and 12 threads per rank yielded identical results to Subsection 5.2.3. The final distribution after the load-balancing was switched off persists. The logs indicate that split-oversized-tree tried to split off the oversized trees but failed repeatedly. Additionally, the secondary strategy seems to be stuck in the inter-rank-distribution state, although all 12 trees on both ranks are fully employed due to the efforts of the primary strategy. This suboptimal behavior demands further investigation. The spread-out's second test-case metrics equal those of Equation 5.8 and Equation 5.10.

Finally, the third test case resulted in an MPI error once again. Multiple ranks wait for an MPI `reduce()`, and the simulation terminates after 1 hour of deadlocking. Logs were only printed during the first second of run-time; no conclusion regarding the workload distribution behavior can be drawn. The error seems to occur mostly on MPI-heavy runs. As the error also occurred during the test of the standalone spread-out strategy, the cascading mustn't be the root of the deadlock. In conclusion, the cascade of spread-out and split-oversized-tree yielded no improvement and provoked an MPI error. This error seems to hold back multiple strategies that use spread-out as a primary strategy. The error's root should be located in the spread-out inter-rank communication.

6. Conclusion and Outlook

This last chapter serves as an overview of the knowledge gained from analyzing the different strategies, evaluating them based on the custom metrics, and testing them under identical conditions to achieve comparability. While recursive bi-partition and split-oversized-tree employ a splitting strategy that triggers top-down splits, the spread-out strategy and its variants use a bottom-up splitting scheme. The top-down splitting is practical if the accuracy is not too important, e.g., in later splitting in a single rank after a good inter-rank distribution has already been achieved. The bottom-up splitting, while achieving great results of equal cell counts in test cases, is useful for initial splitting. While the spread-out-once-grid-stagnates strategy performed well in the tests documented in this thesis, it does not allow scaling.

When referring to all strategies, the spread-out and its cascades can yield MPI deadlocks due to buffer overflows. Additionally, the split-oversized-tree strategy resulted in an arithmetic exception. The last strategy left is the recursive bi-partition. Recursive bi-partition could not fill all trees in the non-MPI run and created over 200 trees in the 2-rank scenario. In the 12-rank scenario, the recursive bi-partition left ranks – and some of their threads – unfilled.

Currently, all load-balancing strategies available to Peano 4, except for the spread-out hierarchical strategy, yield unsatisfying results if the grid is scaled up. The spread-out-hierarchically strategy scored great results in the test case with 2 ranks and 12 threads per rank and scored reasonably well in the extreme intra-rank and inter-rank test cases. Its hierarchical approach uses available compute resources early during grid construction but not from the very start of a simulation. Noteworthy, the spread-out family of load-balancing strategies only distributes initial work and is incapable of AMR-induced necessary run-time rebalancing.

A cascading strategy seems most promising for future approaches to balance loads of Peano 4 or, more generally, any tree-based adaptive mesh refinement software. The spread-out-hierarchically strategy yielded great results for initial workload distribution and parallelizes early grid construction. Afterward, the most promising is a working cascade that switches to a strategy that uses top-down split instructions.

We used the LOH1 scenario as a prime example for the simulated runs, but other applications may result in different load-balancing strategies performing better. The application-wise analysis is recommended. An MPI tracing library like *Score-P* [12] could prove useful for investigating the MPI errors. After the load-balancing code has been reviewed and the errors regarding the MPI and the floating point operation are fixed, starting test runs and using the developed metrics to analyze imbalances across ranks or in ranks before running production code is highly recommended. For this, the metric has to be slightly altered to take maximum thread counts per rank into account.

The metrics that were provided in Chapter 4 have not been used to the fullest in the course of this thesis. Their potential exceeds the analysis of converged load-balancing distributions. Due to the analysis of test cases with static mesh refinement at the beginning of the simulation, the metrics' adapting dynamic properties were not showcased. The metrics can yield insights into balancing behavior over all ranks for the entire run-time.

Additionally, a self-developed functionality not used in the thesis but implemented for load-balancing analysis of AMR simulations is the plotting of local (and remote) unrefined cells over run-time, which can visually convey how the workload shifts in a rank. This feature will also be useful for future load-balancing analysis but was not useful for examining the load-balancing of the test cases with static mesh refinement (cf. Chapter 5). This generated final graphic depicts the local unrefined cells of the single rank 0 in the first, non-MPI, run. Figure 6.1 is focused on the first 12 load-balancing steps, but these graphics can be plotted for an arbitrary temporal scope or the entire run-time. The distance from the blue dots to the red average line give an impression of density around or spread from the average.

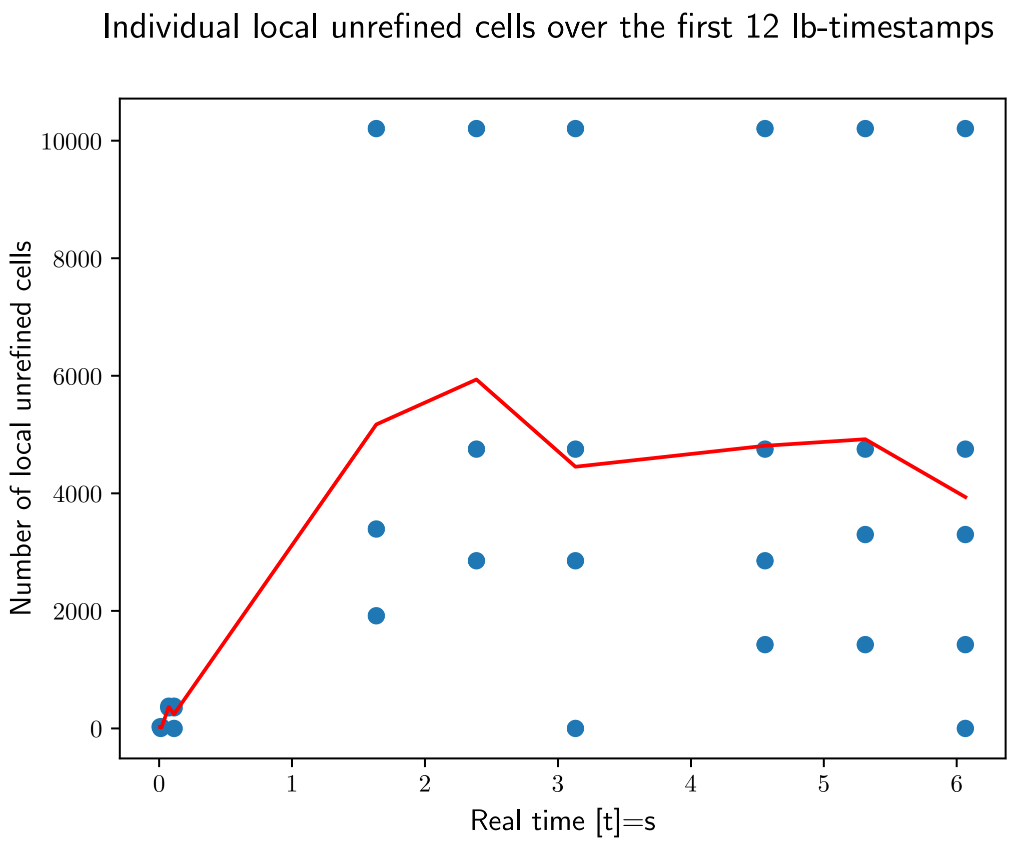

Figure 6.1.: The first 12 load-balancing dumps of the single rank in the non-MPI test case of recursive bi-partition visualized. The red line depicts the average of the timestamp, and the blue scatter points are the local unrefined cells of individual trees per timestep.

A. Appendix

A.1. Inconsistencies of the Source Code with Strategy Behavior

During the Peano 4 source code analysis, two code locations seem to implement unintended behavior. These are noted here and should be investigated in the future.

First, the interaction of recursive bi-partition and split-oversized-tree with their super-class `AbstractLoadBalancing` while querying information about the heaviest spacetree allows ambiguity. Refer to the following two code segments, Listing A.1 and Listing A.2:

```
 1  void toolbox::loadbalancing::strategies::SplitOversizedTree::
        updateLoadBalancing() {
 2
 3    switch (_state) {
 4    case State::InterRankDistribution: {
 5      int    heaviestSpacetree = getIdOfHeaviestLocalSpacetree(_configuration->
            getWorstCaseBalancingRatio(_state));
 6      double weightOfHeaviestSpacetree = getWeightOfHeaviestLocalSpacetree();
 7      //Further logic based on the two variables, omitted
 8    } break;
 9      //Further switch-cases omitted
10    }
11  }
```

Listing A.1: Split-oversized-tree or recursive bi-partiton retrieve the ID and weight of possibly different heaviest spacetrees

In this code segment, the ID of the heaviest spacetree w.r.t. a certain tolerance is retrieved in line 5. In the next function call in line 6, the weight of the heaviest spacetree is retrieved. The problematic behavior is nested inside the call of line 6, responsible for the weight of the heaviest spacetree. For `getIdOfHeaviestLocalSpacetree()`, refer to Listing A.2.

```
1  double toolbox::loadbalancing::AbstractLoadBalancing::
       getWeightOfHeaviestLocalSpacetree() const {
2    const int heaviestSpacetree = getIdOfHeaviestLocalSpacetree();
3    return heaviestSpacetree == NoHeaviestTreeAvailable ? -1 : _costMetrics->
         getCostOfLocalTree(heaviestSpacetree);
4  }
```

Listing A.2: The weight of the unique haviest spacetree is evaluated

In this source code, the second line also retrieves the ID of the heaviest local spacetree and uses it to determine the weight. However, this weight is not taken from a spacetree within a tolerance under the heaviest spacetree, but from the single heaviest spacetree of the rank. Then, the weight is returned. This results in a possible scenario where the heaviest spacetree ID w.r.t. the tolerance determined in Listing A.1 is not linked to the weight that is returned from the true heaviest spacetree of the rank (cf. Listing A.2). The weight of the true heaviest tree is used to further calculate how many cells are split off a tree with a possibly different ID. This is unintended.

Second, the split-oversized-tree strategy seems to restrict the number of off-splits to a potential remote rank by the current ranks free space tree slots. The `computeNumberOfSplits` function is invoked during intra-rank but also inter-rank splitting decisions. Refer to Listing A.3, a direct excerpt from the Peano 4 source code, to trace the behavior.

```
 1  int toolbox::loadbalancing::strategies::SplitOversizedTree::
      computeNumberOfSplits(int sourceTree) const {
 2    int numberOfSplits = static_cast<int>(_costMetrics->getCostOfLocalTree(
        sourceTree) / getTargetTreeCost()) - 1;
 3
 4    numberOfSplits = std::max(1, numberOfSplits);
 5    numberOfSplits = std::min(
 6      numberOfSplits,
 7      _configuration->getMaxLocalTreesPerRank(_state)
 8      - static_cast<int>(peano4::parallel::SpacetreeSet::getInstance().
          getLocalSpacetrees().size())
 9    );
10
11    int currentSourceTreeCells = peano4::parallel::SpacetreeSet::getInstance()
12                                   .getGridStatistics(sourceTree)
13                                   .getNumberOfLocalUnrefinedCells();
14
15    while (currentSourceTreeCells / (numberOfSplits + 1) < _configuration->
        getMinTreeSize(_state) and numberOfSplits > 0)
16    {
17      numberOfSplits--;
18    }
19
20    return numberOfSplits;
21  }
```

Listing A.3: Split-oversized-tree computes the number of trees that shall be split off.

In lines 5 to 9, the minimum is taken from the previous number of splits and the remaining slots from the current rank's spacetree set, l. 8. In summary, the number of splits to a remote rank should not depend on the locally available slots of the spacetree set.

A.2. Parameters for Configuring ExaHyPE

For reproducing the results, the configuration for Peano 4 and ExaSeis, an application of the ExaHyPE 2 framework, are provided:

Peano 4 configuration:

```
python3 configure.py -cc=gcc -cxx=g++ --exahype
                     --multithreading=omp --mpi=mpicxx --cmake=CMAKE
```

ExaSeis configuration:

```
python3 exaseis.py -s ADERDGRusanovGlobalAdaptive --min-levels 3
                   --amr-levels 0 --dg-order 7 -pi 0
                   -lb <load-balancing>
```

The `<load-balancing>` was replaced with the strategy of the according subsections (cf. Section 5.2).

List of Algorithms

List of Figures

List of Tables

Bibliography

[1] Atul Garg. A comparative study of static and dynamic load balancing algorithms. *IJARCSMS*, Volume 2:Page 386–392, 12 2014.

[2] Tobias Weinzierl. The peano software—parallel, automaton-based, dynamically adaptive grid traversals. *ACM Transactions on Mathematical Software*, 45(2):1–41, April 2019. ISSN 1557-7295. doi: 10.1145/3319797. URL `http://dx.doi.org/10.1145/3319797`.

[3] Tobias Weinzierl and Miriam Mehl. Peano—a traversal and storage scheme for octree-like adaptive cartesian multiscale grids. *SIAM Journal on Scientific Computing*, 33(5):2732–2760, 2011. doi: 10.1137/100799071. URL `https://doi.org/10.1137/100799071`.

[4] Anne Reinarz, Dominic E. Charrier, Michael Bader, Luke Bovard, Michael Dumbser, Kenneth Duru, Francesco Fambri, Alice-Agnes Gabriel, Jean-Matthieu Gallard, Sven Köppel, Lukas Krenz, Leonhard Rannabauer, Luciano Rezzolla, Philipp Samfass, Maurizio Tavelli, and Tobias Weinzierl. Exahype: An engine for parallel dynamically adaptive simulations of wave problems. *Computer Physics Communications*, 254: 107251, 2020. ISSN 0010-4655. doi: https://doi.org/10.1016/j.cpc.2020.107251. URL `https://www.sciencedirect.com/science/article/pii/S001046552030076X`.

[5] Tobias Weinzierl. Exahype 2 finally solve some astrophysics problems, 02 2022.

[6] Zhao An, Qilong Feng, Iyad Kanj, and Ge Xia. The complexity of tree partitioning, 2017.

[7] Dominique LaSalle and George Karypis. Multi-threaded graph partitioning. In *Proceedings - IEEE 27th International Parallel and Distributed Processing Symposium, IPDPS 2013*, pages 225–236, 05 2013. ISBN 978-1-4673-6066-1. doi: 10.1109/IPDPS.2013.50.

[8] Zhiling Lan, Valerie E Taylor, and Greg Bryan. Dynamic load balancing for structured adaptive mesh refinement applications. In *International Conference on Parallel Processing, 2001.*, pages 571–579. IEEE, 2001.

[9] Dominic Etienne Charrier, Benjamin Hazelwood, and Tobias Weinzierl. Enclave tasking for dg methods on dynamically adaptive meshes. *SIAM Journal on Scientific Computing*, 42(3):C69–C96, 2020. doi: 10.1137/19M1276194. URL `https://doi.org/10.1137/19M1276194`.

[10] Anju Shukla, Shishir Kumar, and Harikesh Singh. Analysis of effective load balancing techniques in distributed environment. In Kingsley Okoye, editor, *Linked Open Data*, chapter 4. IntechOpen, Rijeka, 2020. doi: 10.5772/intechopen.91460. URL `https://doi.org/10.5772/intechopen.91460`.

[11] Tobias Weinzierl. *A Framework for Parallel PDE Solvers on Multiscale Adaptive Cartesian Grids*. Verlag Dr. Hut GmbH, 01 2009. ISBN 978-3-86853-146-6.

[12] Andreas Knüpfer, Christian Rössel, Dieter an Mey, Scott Biersdorff, Kai Diethelm, Dominic Eschweiler, Markus Geimer, Michael Gerndt, Daniel Lorenz, Allen Malony, Wolfgang E. Nagel, Yury Oleynik, Peter Philippen, Pavel Saviankou, Dirk Schmidl, Sameer Shende, Ronny Tschüter, Michael Wagner, Bert Wesarg, and Felix Wolf. Score-p: A joint performance measurement run-time infrastructure for periscope,scalasca, tau, and vampir. In Holger Brunst, Matthias S. Müller, Wolfgang E. Nagel, and Michael M. Resch, editors, *Tools for High Performance Computing 2011*, pages 79–91, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg. ISBN 978-3-642-31476-6. doi: 10.1007/978-3-642-31476-6_7.