TUM School of Engineering and Design

A Decentral Framework with Dynamic Partitioning for Numerical Computations on Massively Parallel Systems

Christoph Matthias Ertl

Vollständiger Abdruck der von der Technischen Universität München zur Erlangung des akademischen Grades eines

Doktors der Naturwissenschaften

genehmigten Dissertation.

Die Dissertation wurde am 29.06.2022 bei der Technischen Universität München eingereicht und durch die TUM School of Engineering and Design am 10.10.2022 angenommen.

Zusammenfassung

Durch die Nutzung von Hochleistungsrechnen in verschiedenen Bereichen der Wissenschaft ist es möglich immer komplexere Probleme, wie z. B. turbulentes Verhalten in Fluidströmungen oder die Faltung von Proteinen zu bewältigen. Eine der größten Herausforderungen bei numerischen Simulationen von realitätsnahen Phänomenen auf modernen Hochleistungsrechern ist die Partitionierung des Simulationsgebietes. Dazu gehört die Zerlegung des Gebietes in Primitive und deren Verteilung auf alle beteiligten Prozessorelemente. Ziel ist es, eine optimale Balance in Bezug auf die Arbeitslastverteilung zu finden, im Falle eines homogenen Systems eine, die die Last gleichmäßig auf alle Rechenressourcen verteilt, während der Kommunikationsaufwand zwischen den Partitionen so gering wie möglich gehalten wird. Die Situation wird um ein vielfaches komplexer, wenn sich die Zerlegung des Simulationsgebietes während der Laufzeit der Simulation ändert und eine dynamische Neupartitionierung erfordert. Eine Partitionierungsstrategie die während der Laufzeit vielfach ausgeführt wird, muss neben den bisherigen Zielen schnell und die Umverteilungskosten müssen minimal sein.

Erfolgreiche Codes verwenden verschiedene Ansätze um die oben genannte Herausforderung anzugehen. Darunter sind graphen- und geometriebasierte Methoden. Der Nachteil der meisten aktuellen Methoden besteht jedoch darin, dass sie globale Domäneninformationen benötigen. Mit zunehmender Größe und Auflösung der Simulationsdomäne werden sich der Speicherbedarf und der Kommunikationsaufwand in Zukunft als unerschwinglich erweisen.

Der erste Beitrag dieser Arbeit besteht darin, einen umfassenden Überblick über den aktuellen Stand der gängigen Partitionierungsmethoden zu geben. Danach liegt der Schwerpunkt auf der Einführung eines Simulationscodes, der speziell auf ein dynamisches Partitionierungsschema, basierend auf einer lokalen Optimierungsmethodik, zugeschnitten ist. Dieses Framework verzichtet, überall wo möglich, auf global synchronisierte Datenstrukturen. Um den relativ langsamen Fortschritt lokaler Methoden hin zu einer praktikablen Partitionierung zu beschleunigen, kombiniert dieser Simulationscode einen Diffusionsansatz zum Austausch von Arbeitslasten zwischen benachbarten Prozessorlementen mit einer räumlichen und hierarchischen Nachbarschaftsbeziehung, basierend auf einer Baum-Zerlegung des Simulationsgebiets.

Ausgehend von der Prämisse einer vollständig dezentral verteilten Datenstruktur werden alle notwendigen Module für eine numerische Simulation vorgestellt. Dies sind das Kommunikationsmodul, die initiale Partitionierung, das Modul zur adaptiven Netzverfeinerung und -vergröberung, die Lösungsverfahren und die Eingabe/Ausgabe. Besonderes Augenmerk liegt auf dem diffusionsbasierten dynamischen Repartitionierungsansatz.

Schließlich werden drei Testbeispiele mit unterschiedlichen Eigenschaften vorgestellt, um den dezentralen dynamischen Repartitionierungsansatz ausführlich zu testen. Der erste ist ein rein künstlicher Benchmark mit einer sich bewegenden Verfeinerungsfront. Das zweite Beispiel ist einem Benchmark aus der additiven Fertigung mit selektivem Laserschmelzen nachempfunden. Temperaturgradienten werden am Laser sehr hoch und rechtfertigen eine Verfeinerung des Simulationsgebietes. Das letzte Beispiel untersucht eine laminare Umströmung eines Zylinders mit Wirbelablösung hinter dem Hindernis. Das sich ergebende Muster ist auch als "von Kármán Wirbelstraße" bekannt. Um die Strömungsstrukturen genauer zu berechnen wird eine Verfeinerung um Gebiete mit hoher Verwirbelung durchgeführt. Alle Testfälle zeigen gute Resultate des gewählten Ansatzes auf aktuellen Systemen und versprechen eine Anwendbarkeit für zukünftige Hardware und Problemgrößen, bei denen globale Partitionierungsmethoden versagen.

Abstract

The advent of High Performance Computing (HPC) in engineering has enabled the tackling of increasingly complex problems, such as turbulent behaviour in fluid flows or the convolution of proteins. One of the main challenges in numerical computing on modern high performance computers for the simulation of real world phenomena is the partitioning of the domain. This includes the decomposition of the simulation domain into primitives and their distribution among all participating processing elements. The goal is to find an optimal balance in terms of workload distribution, in case of a homogeneous system one that distributes the load equally among all computing resources, while keeping the communication overhead between partitions as small as possible. The situation becomes more involved if the domain decomposition changes during the runtime of the simulation and requires a dynamic repartitioning. A partitioning strategy that has to be executed continuously must be fast and the redistribution costs must be minimal in addition to the previous goals.

Successful codes employ different approaches to tackle the aforementioned challenge. Among them are graph- and geometry-based methods. However, the disadvantage of most current methods is their reliance on globally shared domain information. With an increasing size and resolution of the simulation domain, the memory footprint and communication overhead will therefore prove prohibitive in the future.

The first contribution of this work is to give a comprehensive overview over current state-of-the-art partitioning methods. The main focus thereafter is the introduction of a simulation code especially tailored to a dynamic partitioning scheme based on local improvement methods. This framework renounces data structures, which must be synchronised globally, wherever possible. To alleviate the comparable slow progress of local improvement methods towards a feasible partitioning, this framework combines a diffusion approach to exchange workloads among neighbouring processing elements with a spatial and hierarchical neighbourhood description based on a space-tree domain decomposition.

Based on the premise of a completely decentrally distributed data structure, all necessary modules for a numerical simulation are presented. These are the communication module, the initial partitioning, the adaptive mesh refinement and coarsening module, the solution procedures and the input/output. Special emphasis is laid on the diffusion-based dynamic repartitioning module.

Finally, three test examples with varied properties are presented to extensively test the decentral dynamic repartitioning module. The first is a pure artificial benchmark with a moving refinement front. The second example is modelled after an additive manufacturing benchmark, a laser powder bed fusion on bare metal substrates. Temperature gradients become very high at the laser spot and warrant a refinement of the simulation domain. The last test case is a laminar flow around a cylinder with vortex shedding behind the obstacle. Also known as "von Kármán vortex street". To accurately capture the flow structures, a refinement is performed around regions of high vorticity. All test cases show a viability of the chosen approach on current systems and promise applicability for future machines and problems, where global partitioning methods fail.

Acknowledgements

This thesis was created during my employment at the Chair for Computational Modeling and Simulation (formerly Computation in Engineering) at the Technical University of Munich from 2015 to 2021. I would like to use this opportunity to express my sincerest gratitude to all the people that have helped and supported me to successfully complete this project.

First of all, I would like to thank my PhD supervisor Prof. Dr. rer. nat. Ernst Rank. He has given me the opportunity and granted me great freedom to pursue my interests and has guided me through stimulating discussions on many occasions. In addition, he has created an excellent working atmosphere at the chair, which has greatly facilitated the pursuit of this challenging goal.

Without my mentor PD Dr. rer. nat. habil. Ralf-Peter Mundani this work would not have been possible and I cannot understate his impact on my scientific but also on my personal development. He sparked my interest in computer science and, in particular, in the field of high-performance computing. I am very grateful for his ability to bring order into chaos. Whenever I felt lost or overwhelmed, he helped me chart a path forward. I have also benefited more than once from his love of travel and have always enjoyed our trips together with good conversation and even better food.

I would also like to thank Prof. Dr. rer. nat. habil. Hans-Joachim Bungartz for agreeing to take on the role of examiner for this thesis. In addition, he supported my plan to do a PhD in computer science without hesitation, even though I come from an engineering background. His valuable comments and suggestions were also integral to the successful completion of this thesis.

I have a long history with Prof. Dr.-Ing. habil. Michael Manhart. I got to know him during my bachelor studies when I worked in his fluid mechanics lab and he also accompanied me during my master studies and during the PhD project. I am all the more grateful to him for taking on the role of chairman of my examination committee during the defense of my dissertation.

In the spring of 2020, I was fortunate enough to spend some time at the School of Computational Science and Engineering at the Georgia Institute of Technology, thus I would like to thank Prof. Edmond Chow, PhD for hosting me in his department. Unfortunately, my visit was cut short due to a global pandemic. Nevertheless, I was able to meet with very smart and kind people, which led to many fruitful discussions and I was able to advance my research and gain new perspectives.

Without an appropriate working environment, such a project would of course not be possible. Therefore, I am also indebted to my colleagues and former colleagues. They provided an excellent working atmosphere, both scientifically and personally. Among them are the members of my research group "Efficient Algorithms", Nevena Perović, Bobby Ginting and Jérôme Frisch. With the first two I was lucky to share an office where we had many fruitful discussions and a lot of fun. But also my colleagues from the group "Simulation in Applied Mechanics" at the chair contributed to the fact that I almost always looked forward to coming to the office and they grew very close to my heart. Among them is PD Dr.-Ing. habil. Stefan Kollmannsberger, the group leader with whom I worked closely in teaching and the members of his group (in alphabetical order), Davide D'Angella, Tim Bürchner, Leon Herrmann, Vijaya Holla, Lisa Hug, Philipp Kopp, Nina Korshunova, László Kudela, Oğuz Oztoprak and Alexander Paolini. Last but not least, I would like to give special thanks to Hanne Cornils, our secretary, who took care of all administrative tasks and provided emotional support whenever needed.

I would like to thank my family and especially my parents for their support and belief in me. They created an environment where I always felt safe to pursue my goals and at the same time had a place to which I could always return.

Finally, I would like to thank my partner for her unconditional love and support. She has always believed in me, put her own desires on hold and has shown the utmost patience to bring this project to a successful conclusion. Without her, I could not imagine having finished this work.

Christoph Matthias Ertl
Munich, November 2022

Contents

Chapter 1

Introduction

"A Grand Challenge is a long-term science, engineering, or societal advance, whose realization requires innovative breakthroughs in information technology research and development (IT R&D) and which will help address our country's priorities."
- The 2003 NITRD Program Grand Challenge definition

1.1 Grand Challenges, Applications of High-Performance Computing

In 1991 the United States Congress promulgated the High Performance Computing Act (HPCA) [1], which formally established the High Performance Computing and Communications (HPCC) Program. Within this program, the Committee on Physical, Mathematical, and Engineering Sciences, the Federal Coordinating Council for Science, Engineering, and Technology and the Office of Science and Technology of the United States of America issued a blue book identifying fundamental problems in technology, science and engineering with broad economic and scientific impact, where high-performance computing is needed in order to solve these problems [148]. One year later, in 1992 the participating agencies again issued a blue book revisiting these so-called "Grand Challenges" [205]. After the conclusion of the HPCC Program, the Networking and Information Technology Research and Development (NITRD) Program was established as a successor. Again, within the NITRD Program the "Grand Challenges" have been revisited in 2006 [295]. In its first edition, these challenges included among others:

Computer Science
Machine learning and parallel I/O for other I/O-intensive grand challenges.

Environmental Modeling and Prediction
Large-scale environmental modeling, high-performance computing for land cover dynamics, earthquake ground motion modeling in large basins, massively parallel simulation of large-scale, high-resolution ecosystems and global climate modeling.

Product Design and Process Optimisation
High-capacity atomic-level simulations for the design of materials, first-principles simulation of materials properties.

Space Science
Formation of galaxies and large-scale structures, black hole binaries: coalescence and gravitational radiation, radio synthesis imaging.

Energy
Mathematical combustion modeling, Quantum chromodynamics calculations, Oil reservoir modeling.

In their first issue, the grand challenges have been formulated from the viewpoint of their respective fields. In 2006, the challenges have been reissued. This time, their formulation focuses more on an interdisciplinary viewpoint, where many fields need to come together to solve them. Again, without claim of completeness, current challenges issued by the NITRD Program include:

Knowledge environments for science and engineering
The stated goal here is to make distributed resources such as supercomputers, data archives, distant experimental facilities and domain specific research tools available to a wide audience of researchers, enable scientific progress and establish new fields of science and engineering. Furthermore, this challenge also focuses on the availability of high-performance resources in education to facilitate the training of students in new and emerging technologies.

Clean energy production through improved combustion
This challenge emphasises among others on the ongoing efforts to responsibly use earth's depletable resources. As the still dominant source of energy in the United States, fossil fuels have a major impact on the environment. The goal here is to improve the efficiency of the combustion of fossil fuels to keep the impact on the environment as low as possible as long as new and renewable sources of energy are not able to satisfy the energy demand. To this end, high-performance computing systems that support large and accurate multiphysics applications are needed.

Informed strategic planning for long-term regional climate change
High-resolution regional climate models require massive computing power. One of the most pressing challenges of the twenty-first century is to limit the global warming to 1.5 ° Celsius according to the Parisian climate agreement. To accurately predict and observe the climate and to determine the impact of different influences on the local and global climate, e.g. greenhouse gases or the Gulf Stream, the large distributed climate community needs access to local and global climate data, high-performance computing and visualisation resources.

Real-time detection, assessment, and response to natural or man-made threats
Earthquakes, hurricanes and volcano eruptions are examples of natural threats. Typical examples for man-made threats include chemical, biological and radiological hazards. The challenge here involves the rapid localisation and assessment of those threats to minimise

the loss of life and property. This is a very large topic with many facets. For one, the alleviation of these hazards involve high-performance computing and state-of-the-art models to fast and accurately simulate the outcome of these hazards. The storage, sharing, visualisation and analysation of data is another facet. Furthermore, the development of semi-autonomous robots for hazard removal for example is part of this challenge.

Generating insights from information at your fingertips
The amount of data generated every second is growing exponentially. Everyday devices such as smartphones and cars are fitted with more and more sensors, producing data points in an unprecedented manner. But also in the scientific field the amount of data produced increases manifold each year. Be it data sent by space telescopes like Hubble and James-Web or data produced by the Large Hadron Collider at CERN just to name a few examples. The effort to store these data and to rapidly retrieve accurate insights is an ever growing challenge. Key aspects here involve, to locate information from multiple text sources, archived databases, image archives, and sensor streams for a person or team solving a problem. Furthermore it is imperative to identify and organize connections between disparate pieces of information and finally validate or refute hypotheses and overcome human biases about hypotheses.

It is clear that some of the most pressing challenges of today require more and more computing power. The amount of data in need to be analysed is growing rapidly. The number of natural and man-made hazards in need to be accurately predicted or alleviated is increasing. And, of course many applications in our day to day life need or benefit from an increased availability of computing power. It can be concluded, that in this day and age, high-performance computing is ubiquitous and therefore, many fields of science need to utilise, educate, and advance their use of these resources.

1.2 Massively Parallel Systems

To solve the grand challenges of our time, machines are needed that provide enough computing power to tackle these challenges in reasonable time frames. Machines up for this task, which provide among the highest performance in the world in a single system are called supercomputers. A supercomputer, in comparison to the more common desktop computers, is mainly distinguished not by their hardware, but by an increase in performance of multiple orders of magnitude. The most common way to benchmark the performance of a supercomputer is still the LINPACK benchmark or more recently its portable variant for distributed memory machines HPL [246]. LINPACK itself is actually a software library for numerical linear algebra designed to be used on supercomputers. The benchmark suite was shipped as part of LINPACK's user's manual and has since established itself as the de-facto standard to benchmark supercomputers. As such, it is used in the TOP500 project to rank the worlds most performant supercomputers [301]. The computer this thesis is written on, is a workstation equipped with an Intel Core i7-4790, a microprocessor with four cores, which shows a LINPACK performance of 0.5 GFlop/s. The number one spot on the current TOP500 list as of June 2021 is the supercomputer Fugaku. A machine equipped with almost 160.000 of Fujitsu's A64FX microprocessors, whereas each processor has 48 cores. Hence, the total core count is more than 7.5 million cores. Its listed LINPACK performance is 442, 010 TFlop/s, or in other words 884, 000 times the performance of the above workstation.

One can see, the difference between a supercomputer and a common workstation is not so much a gap in technology. While the microprocessor is certainly more advanced due to the age of the workstation, current workstations may be fitted with the same microprocessors a supercomputer uses. The main difference is the amount of microprocessors and in turn the amount of computational cores a supercomputer incorporates. Looking again at the TOP500 list, the top ten machines comprise core counts between half a million and up to more than ten million. Hence, we rightly call these machines massively parallel computers.

Historically, parallelisation was always one of the main drivers behind an increase in computing performance, but rather than using multiple microprocessors, engineers focused on improvements within a single chip itself. The technological advancement of the microprocessor is often measured by the number of transistors per chip. Since the inception of the *Metal-Oxide-Semiconductor Field-Effect Transistor* (MOSFET), a type of transistor which virtually all modern integrated circuits use, their number has approximately doubled every 18 months. This doubling of the number of transistors was first observed by Gordon Moore in 1965 [221] and later revised in 1975 [222]. Since then, his observation has become known as Moore's Law. More transistors on a single chip translate to an increase in computing power for many reasons. The additional transistors nowadays are used for multi-level caches, whose speed far outperforms external random-access memory [14], wider I/O and memory transfer buffers that allow data transfers per clock cycle by multiples [330], and the ongoing miniaturisation of transistors with tighter integration on a single chip lowers the propagation time of electrical signals, allowing higher clock rates. Furthermore, the power density of transistors stays constant even as they decrease in size, in other words, more smaller transistors require the same amount of power as fewer larger ones occupying the same area. This observation is also known as Dennard or MOSFET scaling [73].

When it comes to parallelisation, the additional transistors are used in multiple ways. The first one being bit-level parallelisation. Bit-level parallelisation refers to operating on multiple bits simultaneously by increasing the processor word size. The earliest computers were only able to operate on a single bit per instruction. An operation on a variable with four bits therefore took four instruction cycles to be completed. Since then, processor word size has increased steadily and was a major factor in the performance increase of early supercomputers [65, 68]. The standard word size has been 64 bits since 2003 with the introduction of x86-64 architectures. So-called vector processors or vector extensions can also be seen as a special form of bit-level parallelisation. Vector processors were the basis of most supercomputers from 1970 to 1990 with notable examples being the Cray-1 in 1975, the Cray X-MP in 1983, the Cray-2 in 1985 and the Cray Y-MP in 1988 [230]. The idea is that execution units perform a single instruction not only on a single variable but on a one dimensional array of variables we generally refer to as vector. As the same instruction is carried out for the whole array in parallel, data dependencies between array elements must be avoided. While vector units are able to greatly improve performance, the application areas are somewhat limited to cases where there are large non-interdependent arrays of variables on which the same instructions have to be performed. This applies for example in numerical simulation or computer graphics [147, 111, 294, 162]. Vector processors have become somewhat obsolete around 1990 when the price for scalar instruction units decreased until it became more beneficial to use more general purpose scalar units instead of one special purpose vector unit, that can only be used for appropriate applications.

On the instruction-level, the added transistors also were used in advancing parallelisation to improve computing performance of microprocessors. We already mentioned scalar instruction units. What makes an instruction unit scalar, is the fact it is able to carry out an instruction in one clock cycle. This is achieved via instruction pipelining. Here, an instruction is divided into stages that operate one after the other, the pipeline. In the classic *Reduced Instruction Set Computer* (RISC) pipeline, the five stages are instruction fetch, instruction decode, execute, memory access and writeback [147]. Very simplified, each stage takes approximately one clock cycle to execute, which results in five clock cycles per instruction. However, with a pipelined architecture, as soon as one instruction has completed a stage, the stage is free to be used for the next instruction. This makes it possible to work on as many instructions simultaneously as there are stages, making it possible to finish an instruction per clock cycle. The additional transistors are used here for the more complex microarchitecture. For example between each stage, registers are needed to store intermediate results. Instruction pipelining was first used in supercomputing from the early 1970s, in machines developed by Seymour Cray for Control Data Corporation and later Cray Research [230].

The next logical step from a scalar processor is a superscalar processor, which is able to execute more than one instruction per clock cycle. This is achieved simply by using more instruction units. These can either be of the same type, for example more than one *Arithmetic Logic Unit* (ALU), or of different types, that would be for example an ALU and a *Floating Point Unit* (FPU). Typically, these units are also pipelined, making the execution of more than one instructions per clock cycles possible [284]. However, with more parallelisation, the underlying problem as well as additional hardware to make efficient use of the additional execution units also play an important role. A program only issuing instructions that use an ALU, cannot benefit from architectures with additional FPUs. Instructions that must be carried out sequentially due to data dependencies also cannot profit from this architecture. The problem of added hardware complexity due to the scheduling hardware needed to efficiently order and dispatch the instruction to their respective execution units has been addressed with *Very Long Instruction Word* (VLIW) architectures, an alternative to superscalar processors. Essentially with VLIW the same instruction units exist to perform multiple instructions in parallel, here however, the number and type of the available instruction units is transient and the the task of grouping parallelisable instructions together is left to the compiler. A single instruction may consist of a multiply and an add instruction for example. The idea behind superscalar architectures has also led to the re-emergence of vector instructions in the late 1990s. Instead of specialised vector processors, a processor may incorporate smaller vector units next to standard ALUs and FPUs to leverage their capabilities for dedicated problems while maintaining applicability for a broader range of problems [147].

As we have seen, the number of transistors per microprocessor had a very large impact on the achievable performance. The miniaturisation of transistors is still an ongoing process with the smallest MOSFETs currently produced being five nanometers. But apart from simply using more transistors efficiently on a chip, another benefit of smaller transistors that was already stated is the constant power density per occupied area. This allowed manufacturers to raise the clock frequency accordingly. As time per clock cycle is the inverse of processor frequency [147], with higher frequencies, the clock cycles become shorter. And with processors able to execute an average number of instructions per clock cycle, a higher frequency directly leads to more instruction per time unit. The practice of increasing a processor's frequency to boost its performance is also known as frequency scaling. Frequency scaling was one of the main aspects behind the performance gains of microprocessors up to the early 2000s. However, two aspects slowed this development down and led to the end of the Moore's Law in its pure form. Firstly, processes able to produce transistors with sizes of three and even two nanometers have been shown to be possible and are being actively developed for commercial use [196, 169, 272]. However this process is likely to stop soon. A single silicon atom is about 0.2 nanometers wide, meaning a two nanometers MOSFET is only 20 silicon atoms wide. At these scales, quantum effects become an issue and prohibit a further decrease in size with the current technology. Secondly, the power consumption in a chip follows the equation $P = C \times V^2 \times F$. P is the power consumption, C is the capacitance, V is voltage and F is the frequency [257]. By increasing the frequency one increases also the power consumption and moreover the thermal power dissipation [137]. The effort to cool the integrated circuits becomes unfeasible after a certain point, so the continued increase in frequency and thus clock cycles per time unit could not continue indefinitely [297].

Instead of faster clock cycles facilitated by the miniaturisation of transistors and an increase in instruction-level parallelism the focus now shifted from enhancing the efficiency of a single core per microprocessor to simply using more cores in a single processor. While frequency scaling came to an end around 2004, when Intel, the world's largest semiconductor manufacturer, shifted their focus on to dual- and multicore microprocessors for commodity hardware [110], Moore's Law essentially kept its' self fulfilling promise of incorporating more and more transistors on an integrated circuit. Having more independent cores incorporated by a single microprocessor gave rise to a new level of parallelism - task or thread-level parallelism. Having multiple cores available allows different tasks to be performed on each core. While each core usually has some exclusive caches, the main memory and in most chipsets some cache levels are also shared between the cores. In this context we also talk about shared memory parallelisation. This form of parallelism puts more effort on the side of the developer to fully utilise the capabilities of the underlying multi-core architecture. A programmer is required to rethink their application in terms of distinct tasks that are able to be carried out in parallel. Naturally not all applications lend themselves to be partitioned into distinct tasks. If tasks rely on an output of another task, the correct order of tasks has to be ensured, even if that means one processor is unable to proceed with its assigned task and has to wait idly if no other tasks are available. Even more fine grained, the programmer is responsible to ensure the correct synchronisation between data variables in the global memory, having to explicitly manage the access to these shared resources.

Actually, task-level parallelism was introduced way earlier than with the introduction of multi-core microprocessors. Neglecting modern technologies like *Simultaneous Multi-Threading* (SMT), a technique to allow multiple threads to be run on a single core, one core within a microprocessor could only execute one task at a time. The logical conclusion to increase performance, apart from increasing the efficiency of the single processor, was to simply use more microprocessors who could work in tandem. The idea of incorporating more processors into one computer was first introduced with the ILLIAC IV, designed from 1960 to 1966. The machine was planned to incorporate 256 processing elements of which finally only 64 were delivered when it finally was completed in 1975. Still, the ILLIAC IV is referred to as the first massively parallel computer [154]. As the design of the computer differed largely from the supercomputers at the time, writing software for the complex system was a difficult task in itself. The findings gained from the effort paved the way for more parallel designs though. In rapid succession, new supercomputers emerged, sporting ever more processing cores. In 1993, in the first year the TOP500 list was published, Fujitsu's Numerical Wind Tunnel marked the top spot [302]. The machine incorporated 166 vector processors, combining the two dominant trends in supercomputing at the time [151]. It's successor as the worlds most powerful supercomputer was the Hitachi SR22021 in 1996 [116]. The system used a distributed memory architecture where every processor operates on its own private memory. In contrast to a shared memory architecture introduced above, the distribution of data to the processing units becomes a major factor in designing efficient applications for these machines. Synchronisation issues on the other hand do not appear, instead data has to be transferred from one processor memory to the next sending explicit messages via a connecting network. Therefore, a fast and reliable high-performance network is crucial for the performance of these machines. As such, the next computers termed the most powerful computers in the world all increased parallelism by using ever more processing elements and connected these elements with more and more efficient networks. The Hitachi SR2201 used a three-dimensional crossbar network, its successor the ASCI Red from 1997 used a two-dimensional mesh to connect its $9,298$ processors [206]. The ASCI Red was also specifically addressed as a message passing computer due to its distributed memory architecture. Other machines with notable networks were from the IBM's Blue Gene Series with the Blue Gene/L using a three-dimensional torus and its successor, the Blue Gene/Q, which used a five-dimensional torus to send messages between its processing elements [4]. An installation of the Blue Gene/L with $131,072$ also occupied the top spot among the world's fastest supercomputers from 2004 until 2008.

Starting in the mid 1990s supercomputers underwent a shift in design philosophy. Instead of building new machines from custom build parts, off the shelf commodity parts were used. One of the first computers that followed this new approach was the Beowulf cluster, build by NASA in 1994 [22]. What is nowadays commonly known as a Beowulf cluster is a pooling of a number of stand-alone computers, connected again with common, off the shelf networking equipment like Ethernet or Infiniband. The typical structure of a cluster is a configuration, where the cluster acts as a single computer which is exposed through a server that communicates with the otherwise invisible compute nodes [67]. On the current TOP500 list, as of June 2021, 92 percent of computers are classified as clusters. The rest, making up 8 percent, are classified as Massively Parallel Processors (MPP). The transitions between a cluster and an MPP are fluid, however, an MPP is generally regarded as more tightly coupled than a cluster, with specifically designed networks. MPP nodes usually are not able to run applications on their own. From this point of view an MPP is much more a single large computer than a cluster [147, 86]. An example would be the aforementioned BlueGene series with it's torus network. However, at the supercomputing level, clusters too have specifically customised network equipment as well as handcrafted software libraries and from the point of view of a user, writing applications for either one is similar.

One of the latest trends in supercomputing, although already roughly twenty years old is adding *General-Purpose Graphics Processing Unit* (GPGPU), so-called accelerators [152, 150]. A GPU in contrast to a classical *Central Processing Unit* (CPU) incorporates many more cores whereas each individual GPU core is much less powerful than a CPU core with reduced clock speeds and instruction sets. GPGPU perform quite well for specialised applications, where rather simple similar tasks need to be carried out over a large set of data without any dependencies and where communication is scarce. Such applications are mainly found in graphics processing, for which GPUs were initially designed, these patterns on the other hand are also found in scientific computing, where the use of GPGPU becomes increasingly popular [256]. As such, seven out of the current top ten supercomputers according to the TOP500 list are fitted with GPU accelerators [301]. Conversely, GPGPUs can also be seen as going the same route as vector processors. They require extensive programming effort to make efficient use of the hardware and are only applicable for specific problems. Therefore, a general-purpose supercomputer, build to work on a variety of problems cannot rely solely on the performance of GPGPU accelerators [133]. For example, the aforementioned fastest computer in the world, the Supercomputer Fugaku does not employ any accelerators and is a pure CPU system.

The trend however, goes to so called heterogeneous systems. A classical CPU is general purpose, meaning it is applicable for a wide range of different workloads and applications with varying patterns of computing, communication and memory access demands. Consequently, more specialised components like the aforementioned vector processors or GPGPUs outperform CPUs in their distinct application areas. Therefore, a heterogeneous system adds a variety of different specialised processing components with different instruction sets on the same microprocessor. Each part of the workload of an application can then be meet with the capabilities of the appropriate component for an additional increase in efficiency. The main challenge in developing heterogeneous systems at the time is not on the hardware side, but in developing appropriate software support to leverage all the capabilities a heterogeneous systems provides [99].

1.3 Partitioning for Massively Parallel Systems

So far, the current challenges in high performance computing have been introduced and a brief overview of the history and the technological advances that have led to the current top supercomputing systems have been given. Currently, the high-performance landscape is made up from a diverse group of massively parallel machines, both memory and message coupled, possibly with various extension units and connected with an array of complex networking hardware in various configurations. The real challenge is to write application code that is able to efficiently use this complex hardware and make use of all the levels of parallelisation provided. The problem to solve must be formulated in such a way, that it can be solved efficiently by a parallel hardware. This entails several key aspects. On the bit- and instruction-levels the hardware and the compiler are mostly responsible for the efficient usage of the available execution units. On the task-level however, it is the developer's burden to conceive suitable subtasks and distribute these among the processing elements. In addition, for message-coupled systems, the necessary communication between processing elements must be specifically laid out.

The Partitioning Problem

In order for every participating processing element to be assigned a share of the problem at hand, the problem must be partitioned into chunks of work. Three general scenarios are common. Firstly, a decomposition of the underlying data of a problem, where the same operation has to be performed on each data partition. Examples here would be a decomposition of the computational domain in physics simulations or the separation of model data to be rendered into an image. Secondly, a partition where individual processing elements are assigned distinct jobs during the solution of the problem, so-called functional decomposition. Many manifestations can be distinguished here, for example a partitioning of the processing elements into management, worker and I/O instances. Further examples involve competitive parallelisation approaches, where the same data is analysed by different solution approaches on different processing elements. Another concept, an assembly line approach similar to instruction pipelining, where chunks of data are processed and transferred from one processing unit to another is imaginable as well. Finally, the last partitioning approach would be a hybrid model where the data is partitioned and all or subsets of the processing elements carry out distinct jobs. For example in complex multi-model climate simulations where different physics have to be simulated on separate chunks of data. Another aspect that greatly influences partition design is the underlying hardware, which the application code should be deployed upon. For example, if a system has specialised execution units, the problems' partitioning should be carried out in such a way that it generates specific tasks which benefit from such a unit. For example, IBM's Blue Gene architecture employs dedicated I/O processing elements to communicate with the file system. Separating I/O tasks and assigning those jobs to their dedicated hardware is most efficient.

When the problem is decomposed, the resulting parts of the problems usually have dependencies between each other. To use the analogy with pipelines again, if a problem is decomposed into stages of a pipeline, each stage is dependent on the results of the preceding stage. An efficient partitioning minimises the amount of communication required between partitions. This is easy for so-called embarrassingly parallel problems, where the individual data points or tasks are mostly independent and show no dependencies. Common embarrassingly parallel problems are for example rendering of computer graphics or convolutional neural networks, where the capabilities of GPGPUs are advantageous. But for the simulation of physical phenomena, where commonly the physical domain is partitioned, the underlying equations are coupled and require the replication and regular update of neighbouring data points. It is crucial to cluster dependent data points within individual partitions to limit the amount of communication between processing elements, as otherwise the overhead of the data transfer hinders an efficient solution process.

Another consideration is the distribution of partitions among the participating processing elements. At first, the computational load should be distributed rather aligned with the capabilities of the underlying hardware. In homogeneous system, an equal work distribution is desirable to be able to use the machine efficiently and without much idling of individual processing elements. In inhomogeneous systems, where computing capabilities are varying, the load must be adjusted accordingly. Furthermore, in systems with special extension units, like the aforementioned GPGPU accelerators, or dedicated I/O nodes, suitable tasks need to be placed here. Of course, this presumes the existence of suitable tasks generated by the partitioning. Another important aspect of task distribution is to limit the impact of communication on the total solution time. That entails to assign partitions that require regular communication among each other on processing elements with favourable positions. In order of their performance either on the same processing element, on processing elements located on the same microprocessor, on the same node or on such elements with the most direct network connection.

An optimal partitioning is almost never trivial and requires much consideration of the application as well as the hardware it should be deployed on. A load distribution where each processing element is used to their full capabilities is often opposing a distribution where a better clustering of data and tasks without fully utilising all processing elements would limit communication overhead. Weighing advantages and disadvantages of different distribution approaches is therefore a major factor for an efficient utilisation of a supercomputer. This task only becomes more involved with the continued trend of machines with more and more diverse processing elements.

For many applications it is sufficient to stop after an initial partitioning and distribution. Each processing element is assigned a share of data and a range of tasks and the application runs on the system until the problem is solved, the simulation timeframe has been exceeded or a critical failure is encountered. This approach is termed static partitioning. Here, the impact of the partitioning on the total simulation time is low, as it has to be carried out only once. As such, an optimal distribution is much more important than faster approaches yielding less than optimal results. Furthermore, the partitioning can be determined on a different machine with different characteristics. For example, partitioning approaches that require complete domain knowledge and therefore a certain size of memory per processing element are possible, even if the target machine does not provide enough. One simply determines suitable partitions for the target machine on a different dedicated machine.

Dynamic Partitioning

There are however also problem classes, where the workload per partition is not static. In so-called *Adaptive Mesh Refinement (and Coarsening)* (AMR) applications for example, the computational mesh representing the simulation domain is able to change regularly to satisfy accuracy requirements given by the solution procedure. A remeshing and a subsequent repartitioning naturally lead to a new distribution of the altered work shares.

Another example could be particle simulations. To compute the forces acting on a particle, other, near particles are taken into account. As particles are able to move throughout the simulation, a partitioning based on particle's physical location of particles must regularly update their assigned particles. Furthermore, there are cases where the computational load of individual tasks is hardly foreseeable at the beginning of an application run. Here, tasks are often dynamically allocated to processing elements whenever they are free. For these applications, partitioning and distribution is a continuous task throughout their runtime.

The overarching goal of a good partitioning is to decrease the time it takes to reach the desired outcome of an application. If the partitioning has to be carried out repeatedly during the runtime of an application, it will directly impact this runtime and a lot more aspects have to be considered in comparison to the static case. Firstly, the quality of a partitioning directly competes with the time it takes to evaluate it. A partitioning with worse quality that is comparably fast to evaluate could be preferable to one with better quality but a higher effort to evaluate. Secondly, partitioning needs to be carried out by the same machine the other parts of the application are computed on. Shifting between machines during a computation is just not feasible, even if only occasionally. This also means partitioning approaches with specific requirements can only be applied if the machine satisfies those requirements. Again, memory requirements are the most common limitations. Partitioning approaches that are only parallelisable in a limited capacity or are even sequential are also rarely feasible, as they leave large amounts of available processing elements idle and waste resources. Finally, the incurred data movement by a repartitioning should be considered. There are schemes, that generate efficient partitions when used in a static approach but lead to large movements if applied continuously. The runtime cost of data migration from one processing element to another has also to be included in the decision process to determine a suitable partitioning scheme.

Extensive effort has gone into the research and development of different methods to partitioning in the context of numerical simulations. [141] gives a comprehensive overview and defines major classes of partitioners. Most methods can be classified into those that divide the computational domain either based on its geometry, that is the location of objects, or by their topology, in other words based on the connectivity or interaction between objects. Later methods are also known as graph-based methods.

1.4 Objectives

The focus of this work is twofold. Firstly, it aims to give an updated overview over different partitioning schemes and their individual advantages and drawbacks. To this end, examples are given from some of the highest scaling application codes deployed on current supercomputing infrastructures from the field of numerical simulations of real-world phenomena. Most of the different partitioning methods have also been implemented in high quality libraries for a broad use. These libraries are referenced wherever applicable. Consequently, this overview aims to serve as a comprehensive analysis of the current state of the art.

As will be discussed later, geometry-based and graph-based methods, especially multilevel methods with global knowledge have been established as the de-facto standard approach to partitioning in the recent past. They are fast, easy to use and provide very good partitioning results. Graph-based methods with only local knowledge are mainly applied as part of a global multilevel method to improve an already existing partitioning. On their own, they have not been considered for some time. To reach a comparable result, local methods take much more time in general. That is because, as their name suggests, they rely only on local updates between processing elements that hold partitions with dependencies among each other.

To achieve their advantageous results the two former methods rely on globally shared domain information. That means, the performance of such methods will always be bound by the effort it takes to synchronise this shared information on all participating processing elements. A lot of the research into these methods has therefore been towards the reduction of global information and the optimisation of collective communication to synchronise the remaining data. These efforts have been very successful, as can be seen by the dominance of such methods. Nevertheless, the trend of ever increasing counts of processing elements to achieve better performance in supercomputing is unabated. More and more processing elements forcibly lead to more partitions to use these processing elements. More partitions also mean more globally shared information and a higher effort to synchronise. Currently the benefits outweigh the disadvantages, but methods that rely on any kind of globally shared information will sooner or later reach their limits. This has also been observed in the recent work of Eibl and Rüde [92], who compared different partitioning approaches for particle dynamics applications. They conclude, although the quality of methods with global knowledge is superior, their largest testcase could only be handled by a local improvement method.

Therefore, the second main contribution of this thesis is the introduction of a code base especially tailored to a dynamic partitioning scheme based on local improvement methods that do not suffer from scalability limits, due to the sharing and synchronisation of global data. To alleviate the comparable slow progress of local improvement methods towards a feasible partitioning, this framework combines a diffusion approach to exchange workloads among neighbouring processing elements with a neighbourhood description based on a space-tree domain decomposition. Space-trees are hierarchical structures found in geometric partitioning methods. They combine a domain decomposition based on geometrical primitives with a hierarchical refinement structure. Defining neighbourhood relations in terms of a hierarchical structure, alleviates the locality of a diffusion method and allows it to quickly find a suitable partitioning over the complete domain. This data structure, in line with the premise of limiting global communication is completely decentralised. In other words, there are no shared structures that need to be synchronised.

The framework in general could be used for a wide variety of applications from the field of numerical simulation. As a prove of concept, the implemented framework uses a finite volume-based spatial discretisation of the computational domain. This discretisation is set up in such a way that it locally degenerates into a finite difference approach. For timestepping an explicit Adams-Bashforth multistep method is used. It is well suited for the numerical simulation of various types of conservation laws. Examples are parabolic differential equations such as the heat equation or the Navier-Stokes equations.

The framework features an adaptive refinement and coarsening module, which allows to alter the domain discretisation over the course of the simulation. Therefore, a dynamic partitioning is advantageous, which repartitions the domain whenever the load distribution deteriorates too much. In addition to the partitioning, all other modules of this framework are subject to the limitations imposed by the decentral design philosophy. Namely the initial decomposition of the computational domain, the solution methods, AMR, and I/O. To this end, the present work details algorithms for each part of the simulation pipeline under the given constraints.

1.5 Outline

The remainder of this work is organized as follows:

Chapter 2: Dynamic Partitioning for Massively Parallel Systems
This chapter aims to give a broad overview over partitioning methods for applications deployed on massively parallel machines. A major classification into geographic and topological methods was already given above. Within this classification, this chapter introduces various manifestations and prominent examples of known applications and libraries, deployed on current supercomputers. As such, this chapter also serves as an overview of the current state-of-the-art.

Chapter 3: A Decentral Framework for Numerical Simulation
In this chapter, the framework concept is introduced. First, the decentral data structure and its primitives are introduced. The neighbourhood model and the communication patterns build upon it are discussed afterwards. Hereinafter, the various modules of the framework are introduced with a special focus on the repercussions of the decentral design on them. These modules include the initial domain generation, adaptive mesh refinement and coarsening, solution methods for systems of linear equations and I/O.

Chapter 4: Dynamic Partitioning
In this chapter the entire dynamic repartitioning scheme is introduced. This includes first the workload model and the communication model. On top of this theoretical framework, the modules that make up the scheme are introduced. These are the diffusion model, the target determination and the actual migration routines.

Chapter 5: Evaluation of the Dynamic Repartitioning
In this chapter, the dynamic repartitioning module is evaluated using three test cases with increasing complexity. The first case is a purely artificial example with a growing sphere around which a refinement is performed. The second example is inspired by an additive manufacturing benchmark test and represents a laser powder bed fusion on bare metal substrates. In the third test case, a laminar flow around a cylinder is simulated, which leads to a flow phenomena known as von Kármán vortex street. The repartitioning scheme is applied to all test cases and evaluated using key characteristics of the distribution. These are the overall workload balance, the connectivity between processes and individual primitives, which gives a measure of the communication overhead, and finally,

the migration requirement to establish the computed partitioning.

Chapter 6: Conclusions
In this chapter a brief summary over the work in this thesis is given. To conclude, possible future research directions are presented.

Chapter 2

Dynamic Partitioning for Massively Parallel Systems

One of the main goals of this thesis is to give a comprehensive overview over current dynamic partitioning schemes used for massively parallel machines. In order to do so, some preliminaries have to be settled. The goals of a successful partitioning are reiterated, then, one has to define primitives to be distributed that make up the individual partitions. The following analysis over several partitioning approaches is based on the work of Hendrickson and Devine [141], Teresco et. al. [299], Schloegel et. al. [279], Bichot [26] and Buluç et. al. [37]. Furthermore, this overview has been updated and extended by recent developments wherever applicable.

2.1 Goals of a Dynamic Partitioning

In most cases, the overarching goal of a successful partitioning is to reduce the overall time an application code needs to reach its desired outcome. This is achieved by decomposing the problem into a set of subproblems that are then distributed among all available resources a parallel machine has to offer. Of course, this presupposes that the problem to be solved benefits from a parallelisation. In general, a partitioning of the problem only makes sense if the problem is large enough. Computing a suitable partitioning, migrating the generated subproblems to their assigned hardware and the added communication between dependent subproblems all add to the runtime of the application. If this added time is larger than gain of using multiple execution units, solving the problem on a parallel machine is not advisable. However, what constitutes "large enough" is dependent on the available hardware and the problem to be solved.

For further discussion, a problem size is assumed that justifies the use of a parallel machine. To reach the overarching goal of reducing the time to solution, four aspects with respect to the partitioning strategy need to be considered:

Balanced and adjusted workloads
It is sensible to adequately involve all available processing elements in the execution of the

program. In addition, the best results are achieved if the workload is distributed such that no processing element is overburdened or idle at any point. In a homogeneous system, in which all processing elements have equal capabilities, an even distribution of workload works best. In an inhomogeneous system, where individual processing elements are stronger than others or have specific strengths and weaknesses, the partitioning approach must be aware of these discrepancies and assign workload accordingly.

Minimal communication between partitions
In so-called embarrassingly parallel problems, partitions have no dependencies between each other. Each processing element is able to work on its assigned share of the problem independently. On the other hand, in the simulation of physical phenomena partitions exhibit data dependencies. These dependencies force processing elements to regularly exchange neighbouring domain values. In general, a good partitioning approach minimises inter-processor dependencies. This goal gains another dimension however, when one takes into account the different speeds of communication between processing elements. Two processing cores located in the same microprocessor communicate faster than two processors on the same node which in turn communicate faster as two processors connected via the network. Depending on the network configuration, there may be further differences in communication speed. A very detailed partitioning might also take these differences in communication speed into account when distributing partitions. That means, partitions with many dependencies and therefore a high communication volume should be assigned to processing elements with the fastest connection possible.

Up until this point the goals are identical to a static partitioning. The difference between a static partitioning and a dynamic one is that the former has to be carried out only once and can be decoupled from the actual application. It may even be processed on an entirely different machine. A dynamic partitioning is carried out continuously during the runtime of the application and therefore has a direct impact on it.

Fast and scalable partitioning
If the partitioning is not static, but should be recomputed regularly during the course of the simulation, the time it takes to evaluate a new partitioning has a direct impact on the overall runtime. As such, the time it takes to compute must be compared to the time a good partitioning saves. An approach that consumes less time with a worse partition configuration is preferable to an optimal partitioning that takes so much time that the better result does not warrant the time investment. Furthermore, the partitioning must be applicable on the target architecture. Pausing the computation, transferring the current domain configuration and workloads to a different machine where an updated partitioning is evaluated and then transferring the results back is simply not feasible to decrease the overall simulation time. Being constraint by the target machines architecture, the approach must be scalable and efficiently run on very large parallel machines. Furthermore, memory constraints have to be considered. Memory per processing element is a rather scarce resource which prohibits approaches which requires complete domain knowledge to evaluate the partitioning. If the number of primitives to distribute becomes large, the memory required to store configurations and supplementing information needed for the partitioning might exceed the memory available.

Minimal redistribution
In a dynamic setting where workloads of partitions continuously change, it makes sense to revaluate the partitioning to account for these changes. To reach the new configuration, primitives must be redistributed. As the cost of the redistribution again has a direct impact on the overall runtime, it is sensible to try to minimise the amount of primitives that have to be migrated.

Unfortunately, these goals are rarely compatible. For example, a configuration with a single partition where all primitives are assigned to one processing element, naturally has a minimal amount of dependencies, namely zero. It would however be the worst possible configuration when it comes to balanced workloads as only one single processing element would be working and all others are wasted. It has been shown that finding an optimal solution between these two goals is NP-complete [299, 122, 121]. In a dynamic setting, where two more goals have to be sufficiently satisfied, the problem becomes even more involved and it is unpractical to look for an optimal solution. Therefore, existing partitioning methods use heuristics which provide different compromises between these goals.

2.2 Granularity and Workload

The scope of this thesis is restricted to numerical simulations of physical phenomena that can be represented by a mathematical model. A mathematical model usually is a coupled differential equation, representing the relations between functions of physical quantities and their derivatives representing their rates of change. For example, differential equations are used to represent phenomena from fluid and solid mechanics, thermodynamics and sound propagation. After a suitable model has been derived for the problem at hand, the second step is to transfer the continuous model into a discrete representation that can be evaluated by a computer. This step is called discretisation accordingly. Discretisation techniques themselves are further divided into mesh-based methods and mesh-free methods.

Mesh based-methods define a set of points and their connectivity, the mesh. At the mesh points the unknowns of the underlying function are evaluated. The most widely used mesh-based methods are the Finite Difference Method (FDM), the Finite Element Method (FEM) and the Finite Volume Method (FVM). In the FDM the function values are evaluated at the mesh points directly, derivatives are approximated using the values at neighbouring mesh points. In the FEM, the domain is separated into a set of elements on which basis functions are defined. The linear combination of these basis functions approximates the solution function which is recovered by solving for the unknown coefficients of the basis functions. In the FVM a control volume around each mesh point is defined. The basis used here are the volume integral equations that describe the quantities within each control volume. These are then transformed to surface integrals and their compatibility across volume surfaces is required. Mesh free methods on the other hand still evaluate quantities at specified points in space and time. Here, the points do not have to adhere to a static connectivity to their neighbours and are allowed to move from one time to another. One of the oldest mesh free methods, smoothed-particle hydrodynamics (SPH) defines points as physical particles that hold density and mass information for example. Regardless of the discretisation technique used, all of them generate a system of coupled algebraic or polynomial equations that can be solved on a computer.

For solving this system of equations on a parallel machine it is decomposed based on the unknowns at the points defined by the discretisation. Each subsystem represents a task. The size of these tasks, and therefore their computational load determines the granularity. A fine-grained decomposition only incorporates a few mesh points or particles per task, a coarse-grained decomposition contains a large amount per task. The granularity is represented as a measure of computational work per task [163]. The granularity directly influences the partitioning. A fine-grained decomposition is preferable to reach a balanced workload. A lot of small tasks can be freely distributed among the processing elements such that a good balance is reached. On the other hand, the large amount of tasks introduces a large bookkeeping overhead and the number of inputs to consider for the partitioning is high as well. A coarse-grained partitioning has exactly the opposite advantages and drawbacks. Fewer, larger tasks make it harder to find a suitable workload balance. In the worst case, exactly as many task as processing elements exist and there is no distribution that adequately serves the capabilities of the individual processing elements. However, finding a partitioning and keeping track of the small number of tasks requires a comparably low amount of effort. Finally, the granularity of the decomposition also may have an impact on cache efficiency. To this end, a lot of research has been conducted to determine the granularity for an increase in performance. The interested reader is referred to [56, 339, 209, 190, 223]

Another important aspect of the granularity is the workload metric it is based on. A partitioning requires not only the amount of tasks but also a somewhat accurate measure of each tasks' workload to distribute them to processing elements in a balanced fashion. Watts and Taylor state that an *accurate load evaluation is necessary to determine that a load imbalance exists, to calculate how much work should be transferred to alleviate that imbalance and, ultimately, to determine which tasks best fit the work transfer* [323]. Although they state this in the context of a runtime load balancing, that is a rebalancing when tasks are already distributed, the same notion holds true for the initial distribution of workloads.

Workload can be estimated in two ways. An a priori estimation uses knowledge of the model-specific algorithms. For example the number of mesh points within a partition. If the number of mesh points changes due to an adaption in an AMR application, the added, or subtracted workload can be estimated and the imbalance can be remedied by repartitioning, i.e. migrating mesh points to regain a balanced amount among processing elements. This approach becomes more difficult, the more complex the model itself becomes. If for example not every mesh point translates to the same workload depending on various known factors like the amount of data dependencies it has and on various unknown factors like the amount of iterations it takes to reach a defined minimal error measure in a linear solver. Furthermore, the system the application is running on itself skews the load estimate. Again, there are factors that are known and can be taken into account like cache sizes, but there are also unknowns, like load from simultaneously running applications, which is common for shared systems. Commonly used a priori load indices can be found in [231, 335, 100].

Consequently, the second way of estimating future workload is to simply take a posteriori measurements and extrapolate these measurements under the assumption that load will only slightly change. Measurements are usually taken directly in runtime or executed clock cycles. The applicability of this method for load balancing is limited though, because the assumption of a minor variation breaks down when the computational domain is adapted. Furthermore, for initial distribution of tasks an a priori work load estimate is at least required once. Nevertheless, an a posteriori load measure is still very much warranted in combination with an a priori one to check the accuracy of the later and inform the user of too large discrepancies between estimated and measured workloads.

2.3 Geometric Methods

Geometric Methods are the first major class of partitioners. Again, the task of a partitioner is to take tasks or objects and map them to the available processing elements. As input, every partitioner needs to be aware of the workload or weight of each object. In addition, geometric partitions rely on the geometric location of objects to form partitions. In other words, they try to cluster geometrically near objects into partitions of equal workload. This is especially beneficial for applications, where the main interaction between objects come from those that are physically close. That is the case for contact-based phenomena like crash simulations, where a large part of the workload comes from searching for colliding objects. When all physically close objects reside on the same processing element, the search for colliding objects could only be performed with local objects. Another example where geometric methods perform well are particle-based simulations like smoothed-particle hydrodynamics or molecular dynamics. Here, particles interact only with physically near neighbours. Again, if all neighbours are assigned to the same processing element, these interactions can be computed very fast.

In general, geometric methods are conceptually simple compared to topological methods. As such, partitioning using a geometric method is usually quite fast. Hence, geometric methods are also often the first choice in highly dynamic simulation scenarios where a frequent repartitioning is warranted. A further advantage of geometric methods is their independence from information about the dependencies of objects. Especially for mesh-free discretisations, generating this information might be expensive. On the other hand, the lack of this informations does not allow for an explicit control of communication costs between partitions. As stated, these methods operate under the premise that most dependencies are geometric and are sufficiently dealt with by design. Therefore, the partition quality is considered worse in general compared to topological methods that are communication aware [76, 29]. Finally, geometric methods may generate disconnected subdomains for complex geometries. Another side effect of the lack of dependency information.

2.3.1 Recursive Bisection

Among the simplest partitioning approaches are recursive bisection methods. The basic idea is to take a line in two dimensions or a plane in three dimensions and cut the computational domain along them into two equal partitions. This division is then recursively applied to the new partitions until the desired number of partitions is generated.

Berger and Bokhari where the first to propose such an algorithm, based on a bisection along the coordinate axes [23]. As all geometric partitioners, the algorithm takes the geometric locations of all primitives, as well as their weights and tries to find a cutting plane that splits the total weight of all objects into two equal sets. As it recursively splits the objects along a coordinate axis, it is aptly named Recursive Coordinate Bisection (RCB). This method is very simple to implement, fast and produces good results [300]. However, the method has two drawbacks. Firstly, it only produces a number of partitions which is a power of two. Secondly, for well shaped meshes, the use of a line to divide the computational domain is able to also keep the amount of communication small as long as the ratio between the smallest and largest mesh element is bounded [46]. For RCB that is unfortunately not the case as the method only considers the weight of the objects and not their density, in other words the number of objects per partition. It therefore may produce halves of equal weight, but vastly different amounts of objects [216, 300]. To remedy this, another approach was proposed by Jones and Plassmann called Unbalanced Recursive Bisection (URB) [172]. Here, the cutting plane is chosen such that it divides the number of objects in half also considering the aspect ratio of the newly generated partitions. Furthermore, partitions are only further divided proportionally to the total object weight within them, resulting in a balanced partitioning on one hand, and an arbitrary number of partitions on the other.

Both RCB and URB are incremental methods, that means a small change in the underlying problem only induces a small change in the decomposition [141]. As such, they display minimal costs when it comes to the redistribution of objects, which is a beneficial property for the use in a dynamic scenario. Therefore, they have been successfully applied to problems from fluid dynamics [216], molecular dynamics [292] and transient dynamics, in particular for contact detection and smoothed-particle hydrodynamics [287, 251]. Furthermore, RCB has been applied for various other particle-based applications [108, 109] and for the discrete element method [61, 62].

Another variant that uses the idea of recursively dividing the computational domain with lines or planes is the Recursive Inertial Bisection (RIB) proposed by Simon [289, 298]. In RCB and URB, the cutting planes are chosen to be orthogonal to the coordinate axes. The idea of RIB is to use the axes of principle inertia instead of the coordinate axes to bisect the computational domain. This comes from rigid body dynamics, where for a general three-dimensional body one can find three orthogonal axes, for which the products of inertia are zero. These are called axes of principal inertia. In rigid body dynamics, it is preferable to formulate the problem in the coordinate system spanned by these axes. In RIB the objects and their weights are treated as a rigid body to calculate its axes of principal inertia. Unfortunately, it is not trivial finding these, as one has to solve a characteristic value problem. The three eigenvalues then give the three sought after axes. RIB yields better quality partitions than RCB and URB but can not compete with more sophisticated graph partitioners in general [300, 98]. In contrast to the former two methods, RIB is not incremental [106] and it scales moderately [300], which limits its applicability in dynamic scenarios.

The last approach from the class of recursive bisection partitioners comes from Miller et al. [215, 214]. They propose an algorithm, where they project all n-dimensional objects to a random n+1 dimensional sphere. The cutting plane to bisect computational domain is then placed through the center of the sphere. In theory, this method gives the best results of all recursive bisection methods, even comparable to graph partitioners. But is is the most complex and expensive variant of the bisection methods [125].

As recursive bisection partitioners are conceptually very simple and easily parallelisable [29, 285, 140], they are often implemented by application developers directly into their codebase. Nevertheless, these algorithms have also been implemented into the high level libraries Meshpart [124] for the Matlab environment and into Zoltan [30], a library with a wide range of different partitioning methods for C and C++ applications. The implementations in Zoltan are parallelised using the Message Passing Interface (MPI) [225]. An application of the Zoltan implementation of the RCB for the simulation of a hopper discharge using the discrete element method can be found in [204].

2.3.2 Space-Filling Curves

Another method from the class of geometric partitioners make use of Space-Filling Curves (SFC). SFCs map a multidimensional space on to a linear one dimensional array while locality information is preserved to a certain degree. In other words, objects that where close in their original space will also be close in the one dimensional mapping. The array can then be cut into a number of continuous partitions, adjusted to the number of processing elements and their capabilities. Such a partitioning inherently leads to moderate communication costs due to the locality information present in the curve [91]. There exist various SFCs, all with their individual advantages and drawbacks. The first SFC was discovered by Peano in 1890 [241]. The most relevant SFCs for dynamic partitioning are the Hilbert curve [149], the Lebesgue curve [193] and the Sierpiński curve [288]. A detailed overview over the different SFCs, their mathematical properties and their applications in computer science can be found in [10] and [265].

To map objects with multidimensional coordinates to one dimensional space, each object is given an index that represents the object's position along the SFC. The objects are then sorted by their index [5]. [20, 75, 30] describe binning techniques to evaluate this index. Usually though, SFCs are typically generated by a two step process.

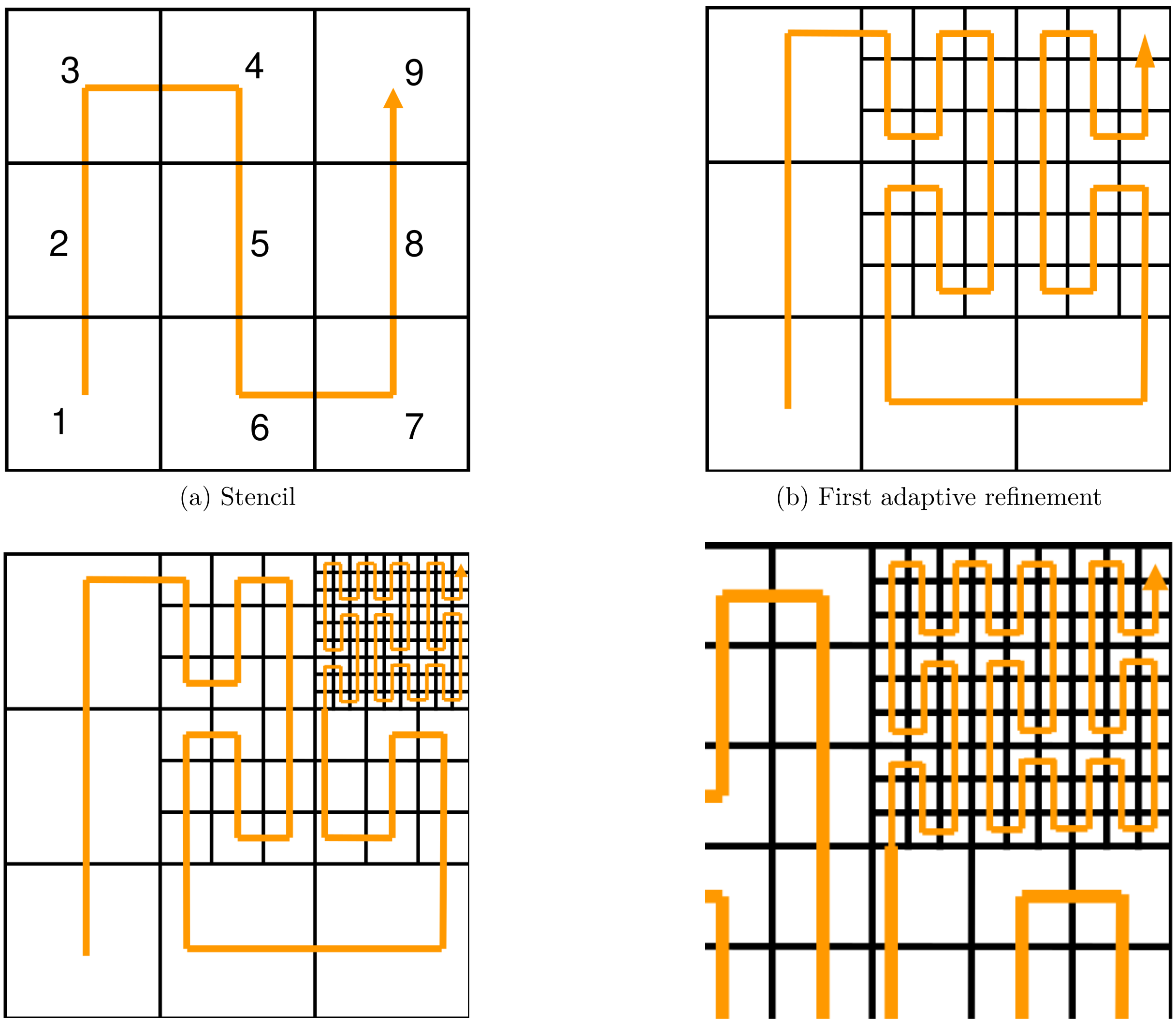
First the computational domain is recursively refined into spatial subdomains, whereas each subdomain contains no more than a single object in the end. One starts with a single root node representing the entire space. If the space has more than one object in it, it is refined into a defined number of subdomains. This is then applied continuously until each subdomain has zero or one object in it. The amount of subdomains generated for each refinement is dependent on the SFC used. Both Hilbert and Lebesgue curves bisect the domain in each cardinal direction per refinement. The resulting tree, called quadtree in two dimensions, because it generates four children for every refined parent node and octree in three dimensions respectively, are well known structures in computer science. For the Peano Curve, the domain is trisected in each cardinal direction, resulting in nine children nodes for every refined parent node in two dimensions and 27 children nodes in three dimensions. To construct a Sierpiński curve, the domain is divided into regular triangles in two dimensions and tetrahedral elements in three dimensions respectively.

The leaf nodes of the resulting refinement tree are then traversed in a specific order given by the SFC [45, 106, 105, 107, 104, 217, 69]. Each SFC defines a stencil that describes the order in which each subdomain is visited. This stencil is applied to each level of the refinement tree. Therefore, SFCs are self-similar. Appropriate similarity transformations allow the connection of open ends.

In general, SFCs can be computed very fast, comparable to the runtimes exhibited by RCB [249, 250]. Furthermore, partitioning with an SFC is incremental, the redistribution costs of a repartitioning in a dynamic setting are low. From an implementation perspective, SFC partitioners range between the simpler geometric methods and the more sophisticated graph partitioners in complexity [91, 236]. Tree data structures and especially octrees are also widely used for mesh generation and adaptive mesh refinement [13, 77, 141, 286, 226, 211, 293]. If the domain is generated following a tree refinement, they are especially well tailored for an SFC partitioning, because the first step, the generation of the tree is already completed.

In Figure 2.1a the stencil for the Peano curve is illustrated in two dimensions. In Figure 2.1b the domain is adaptively refined by one level. As described above, the curve visits all subdomains in the order defined by the stencil. It is applied independently to every refinement level. As such, the path through the refined domain is generated by applying the stencil on the initial coarser representation and then again locally applying it again to each refined subdomain individually. The Peano curve is not unambiguous, meaning the stencil can be applied in different ways, which results in different paths through the domain. Figure 2.1c and Figure 2.1d illustrate another adaptive refinement and a possible resulting curve. The later illustrates a detailed view of the right upper part of the domain. The Peano Curve is used for example in the appropriately named Peano Framework [324, 325, 38], a solver framework for partial differential equations. The Peano Framework is the basis behind ExaHyPE, an engine to solve hyperbolic partial differential equations specifically tailored to efficiently use future exascale parallel machines. It addresses grand challenges in geo- and astrophysics, such as the dynamic rupture processes and subsequent regional seismic wave propagation [261]. Furthermore, ExaHyPE is one of the codes used within the Center of Excellence in the domain of Solid Earth (Cheese) project that addresses challenges in computational seismology, physical volcanology and tsunami modeling [55].

The Hilbert curve is one of the most used SFCs in literature as it is based on traversing the widely used quad and octree data structures. Its two dimensional stencil is shown in Figure 2.2a. Figures 2.2b and 2.2c show consecutive adaptive refinements of the domain and possible traversals gained by applying the stencil. Similar to the Peano Curve, there



(a) Stencil

(b) First adaptive refinement

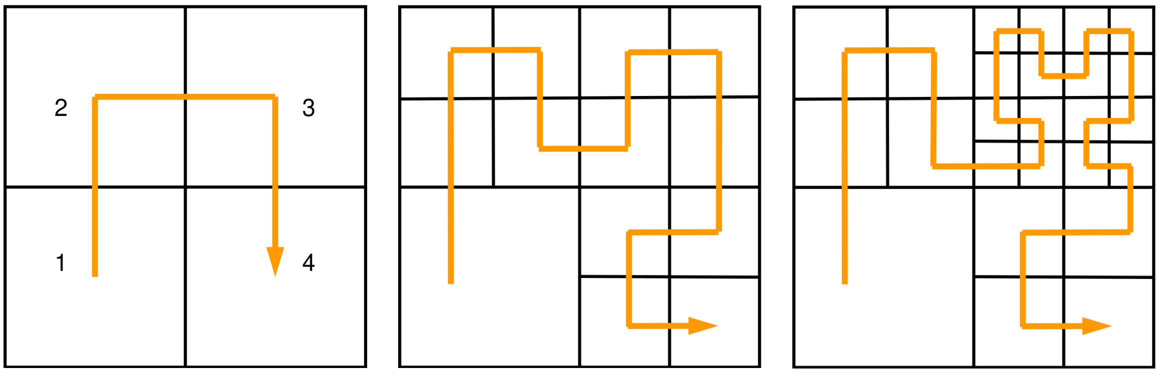
(c) Second adaptive refinement

(d) Detailed view of the second adaptive refinement

Figure 2.1: Peano Space-Filling Curve

are different possibilities for applying the stencil for the Hilbert curve and there exist alternative configurations. Among the earliest applications for these curves are particle-based gravitational simulations and smoothed-hydrodynamics [321, 249] and the decomposition and ordering of finite element meshes [91, 240, 235, 232]. The Zoltan package also provides a partitioning tool using the Hilbert curve [30]. A more recent example is SeiSol, a code for the simulation of seismic wave propagation and earthquake scenarios and another code from the Center of Excellence in the domain of Solid Earth project [283]. SeiSol also uses Hilbert SFC-based partitioning [262] to achieve competitive performance on current petascale architecture [34, 308].

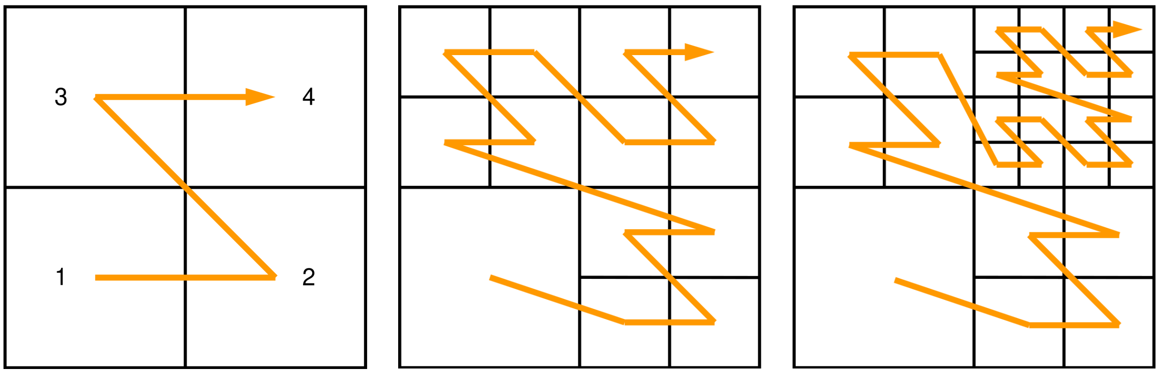
The second curve that can be inferred from quad and octree data structures is the Lebesgue curve. The curve conforms to a depth-first traversal of the refinement tree. Its stencil is shown in Figure 2.3a in two dimensions. One can observe the characteristic



(a) Stencil (b) First adaptive refinement (c) Second adaptive refinement

Figure 2.2: Hilbert Space-Filling Curve

"Z" shape of the curve for which it is also called Z-order curve. Figures 2.3b and 2.3c show the curve after one and two adaptive refinements of the domain respectively. The Lebesgue curve does not need similarity transformation to allow the connection of open ends, therefore there is only one canonical configuration of the curve. When compared to the Hilbert curve, it does not persevere locality as well. This is due to the large jumps in the linearisation which can be seen in the Figures 2.3 in the transitions from quadrant two to quadrant three. This is even more pronounced in three dimensions [45, 167]. However, the Lebesgue curve has one very important advantage over the Hilbert curve. The computation of the index can be very efficiently carried out by interleaving the bit-wise representation of the coordinates of an object, introduced by Morton [224].



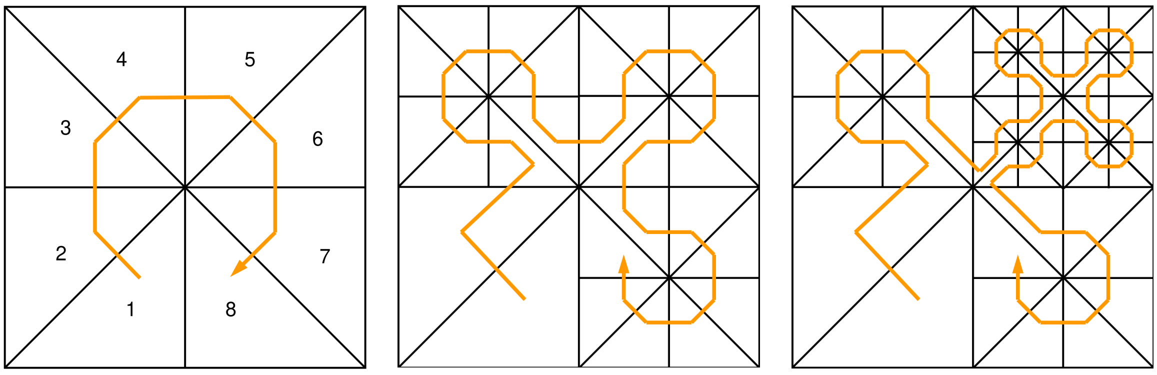
(a) Stencil (b) First adaptive refinement (c) Second adaptive refinement

Figure 2.3: Lebesgue Space-Filling Curve

The first use of a Lebesgue curve for partitioning was reported by Warren and Salmon for particle-based gravitational simulations [321]. More recently Frisch used an octree-based data structure with a Lebesgue SFC-based partitioning for a computational fluid dynamics framework for the simulation of large-scale fluid mechanics phenomena and thermal comfort assessment [113]. The code was specifically tailored for massively parallel computers and has shown competitive performance on current supercomputers [115].

Furthermore, the Lebesgue curve is used in the high-level library for parallel adaptive mesh refinement p4est. p4est uses a connected octree data structure, termed forests and implements various highly optimised algorithms for domain management and partitioning [42, 165, 39]. p4est has been used as a basis for ParFlow, a hydrology modeling software that simulates saturated and variably saturated flow in heterogeneous porous media [237]. Using p4est, ParFlow is able to scale close to half a million processes on a BlueGene architecture [40, 41]. Another high-performance framework waLBerla, for the solution of fluid dynamics and particle dynamics problems, also employs an interface to p4est and is therefore able to utilize their mesh management and partitioning capabilities [21, 280].

The last SFC used for partitioning is the Sierpiński curve. It is used to traverse a domain decomposed into regular triangles in two dimensions and tetrahedra in three dimensions. In Figure 2.4a the stencil of the curve in two dimensions is shown. Figures 2.4b and 2.4c show possible configurations for one and two adaptive refinements respectively. The curve has been used for adaptive mesh refinement applications [12], finite element and finite volume methods [211] and for the solution of shallow water equations [11]. The code framework sam(oa)^2 for the simulation of tsunami wave propagation also uses Sierpiński curves to partition the computational domain and allows the code to perform on current top machines [268].



(a) Stencil (b) First adaptive refinement (c) Second adaptive refinement

Figure 2.4: Sierpiński Space-Filling Curve

2.4 Graph-Based Methods

The second major class of partitioners are graph-based methods. Graph-based methods view the domain decomposition as a weighted graph, where the nodes represent the objects and the edges represent dependencies between the objects. The weight of the nodes represents the workload incurred by the object, the weight of the edges represents the cost of the communication to satisfy these dependencies, in other words, the amount of information required to be transferred. In contrast to geometric partitioners, graph-based methods hence model the communication but neglect the location of the objects.

The graph partitioning problem in general defines an objective function that maximises or minimises an objective. One of the most used objective functions is to minimise the total edge cut, in other words to minimise the total amount of cut edges when assigning nodes to a partition. When applying the graph partitioning problem to assign objects to processing elements of a parallel machine, a better objective function is to minimise the total communication volume [142]. That means the goal is to minimise the total weight of the edges cut by the partitioning. In the context of partitioning for parallel machines, where the task is to generate a mapping which assigns objects to processing elements an additional requirement is to balance the workloads applied by the objects among the processing elements. The problem therefore is augmented by a balance constraint which states that each partition's workload must be equal, except for an accepted imbalance.

The graph partitioning problem and the constraint graph partitioning problem can be applied to a wide range of different applications where the physical or technical phenomenon can be described by a graph. Buluç et. al. [37] give a wide variety of applications where the partitioning of a graph is useful. For example it can be applied for general graph problems such as eigenvalue computations [31] and database management [266]. Moreover graphs are present wherever a network is present, therefore complex networks such as power grids [85, 199], biological networks [173], social networks [112] and road networks [192, 218, 184, 70, 71, 202] can be represented as a graph and graph-based partitioning can be applied. Finally, there are also applications in scientific visualisation and image processing, for example in volume rendering [9] and image segmentation [44, 244].

There exist a wide variety of algorithms that provide an optimal solution for the general and the constraint graph partitioning problem. Unfortunately, these algorithms are NP-complete [122, 121], which means their runtime complexity is exponential to the number of inputs. In other words, a graph with more than a few hundred nodes can not be partitioned in any reasonable amount of time. That prohibits their use for static partitioning and for dynamic partitioning, where the speed of the partitioning is crucial, even more. As such, this overview is limited on heuristics which provide reasonable good approximations to an optimal partitioning in just a fraction of the time. For a more comprehensive overview including also the exact algorithms, the interested reader is referred to Buluç et. al. [37] who gives a more general overview over graph-based methods.

2.4.1 Global Graph-Based Methods

Graph-based methods are further classified into global and local methods. Global methods work with the complete graph while local methods can be applied to arbitrary connected subsets of a graph. With the complete information about the graph, global methods are able to compute a solution directly, but they are mostly restricted to smaller graphs or used as static partitioners because of their comparably high cost. In this section, Graph Growing 2.4.1.1, Node-Swapping 2.4.1.2 and Recursive Spectral Bisection 2.4.1.3 are discussed. Multilevel methods are a special kind of global graph-based methods that allow the combination of several global and local methods, therefore, they are discussed separately in their own section 2.4.3.

2.4.1.1 Graph Growing

One of the simplest forms of global graph partitioning is graph growing, also called greedy graph partitioning [299] or levelised nested dissection [279] in literature. The algorithm starts with a single node and adds connected nodes sequentially to the current partition until the desired partition size is reached [123, 97]. New nodes are usually selected using a local search through all connected candidate nodes with the goal of minimising the edge cut weight when adding a node [178]. The quality of the generated partitions are highly dependent on the chosen starting node. As such, a common strategy is to run the algorithm many times over with different starting nodes and selecting the outcome with the best properties. As the algorithm does not include any geometry information, it does not maintain geometric locality [316]. Furthermore, the algorithm tends to leave many separated nodes for the last partitions which leads to fractured partitions [160]. The method on its own is not incremental and not easily parallelisable, which limits its applicability for dynamic partitioning. Nevertheless, it is very fast [98, 310, 316, 320] and produces partitions comparable to RIB in quality [98]. A broader overview over different variations and improvements of the concept can be found in [60, 127, 263].

Graph growing has also been extended to start from more than one node to generate multiple partitions at once. This so-called bubble framework is based on the ideas of [320] and has been first examined in [82]. After selecting a set of evenly distributed starting nodes, graphs are simultaneously generated from each node. After all nodes are assigned to a partition, new starting nodes are selected for the next trial. Improvements to this approach have been examined in [274, 213, 212], which range from different growing methods to extensions for weighted graphs.

2.4.1.2 Node-Swapping

Node-swapping is another method for global graph-based partitioning. The idea comes from partition refinement where one tries to improve the partition quality of an already existing partitioning. Using a random partition as a starting point, allows to use the schemes from this field to be used as a partitioner in itself. The basic concept can be explained as follows: given a starting partitioning with two partitions, one finds two subsets of equal size and swaps these subsets from one partition to the other. The subsets must have equal size as to not violate the balance constraint. The goal is to find the subsets from each partition such that swapping them yields the greatest reduction in edge cut or edge weight respectively. This swapping of subsets can be repeated until no improvement is possible anymore. Finding those subsets to swap optimally is not trivial though.

Therefore, Kernighan and Lin [183] proposed an algorithm that repeatedly swaps only single node pairs. The algorithm introduces a queue for every partition and sorts the nodes within the partition according to their priority, that is the decrease in edge cut weight a transfer to the other partition yields. This decrease might also be negative. After sorting, the nodes with the highest priorities from each queue are swapped and removed from the queues. After both queues are empty and all nodes have been swapped, the node swapping is reversed to the point at which the total edge cut weight was minimal.

The resulting partition is then used as a starting point and the algorithm is applied again, filling the queues, swapping nodes and reversing to the minimum edge cut of the current pass. This is repeated until a complete pass through the queues yields no improvement any more. The method also termed KL refinement, unfortunately is rather expensive with a quadratic complexity per pass. Although some improvements have been proposed to reduce its complexity [90].

Another variation was proposed by Fiduccia an Mattheyses [102], termed FM refinement accordingly. Here, instead of swapping node pairs, only a single node is transferred from one partition to the other at a time and the priority queues for each partition are updated immediately. The main advantage of the FM refinement over the KL refinement is its linear complexity. While both KM and FM schemes are able to escape some local minima, this ability is limited. Therefore, the quality of the final partitioning is highly dependent on the initial configuration of the partitions [279]. One way to improve the quality is to follow a similar approach as graph growing and simply run the method for many different starting configurations and pick the best outcome.

Other alternatives to improve the quality of these partitioning methods have been proposed. Among them, allowing only boundary nodes to be swapped, halting a pass through the priority queues early [179] and very localised versions where the search for matching pairs only considers directly connected nodes [233, 269]. Furthermore, node-swapping methods work better if the balance constraint is more lenient, allowing them a greater freedom in moving nodes from one partition to the other [36]. This is used by Holtgrewe et. al. [153], Sanders and Schulz [271, 282] who allow trade-offs between node balance and total edge cut weight to generate partitions with higher quality. Finally, the problem of finding larger sets has also seen some progress. Inspired by theoretical considerations made by Hromkovič and Monien [159] several authors have proposed ways to find larger sets of nodes to exchange in a single step [80, 8, 135, 220]

In summary, node-swapping methods produce medium to high quality partitions but are relatively expensive. They rely on global information and are paralleliseable only in a limited capacity [279, 126]. Finally, they are not incremental. As such, these methods are mainly used for static partitioning once at the beginning of an application when used on their own.

2.4.1.3 Recursive Spectral Bisection

Similar to recursive bisection methods using geometric information, a similar method can also be constructed using the connectivity information present in graphs. To bisect the nodes into two distinct sets, one computes the Laplacian matrix of the graph where the off diagonal elements represent the connectivity between nodes, similar to the adjacency matrix and the diagonal elements represent the degree of the node, that is the number of connections the node possesses [59].

The second smallest eigenvalue (the smallest eigenvalue is zero), gives a measure of the overall connectivity of the graph. A graph that is not connected has a value of zero in this eigenvalue. The eigenvector associated with this eigenvalue is called Fiedler vector after

Miroslav Fiedler who laid the theoretical foundations of the method [103]. The Fiedler vector associated with each node then gives a measure of the distance between the nodes, which is used to sort the nodes and bisect the sorted list of nodes into two halves. Again, this method is recursively applied on the newly generated subsets to generate the desired number of partitions. First use of this method has been reported by Donath and Hoffman [83, 84], Pothen and Simon et. al. [252, 289, 253] and Boppana [32].

Recursive Spectral Bisection (RSB) gives very high quality partitions, unfortunately computing the eigenvectors is very expensive and the overall algorithm is very slow [279, 320]. In practice a modified Lanczos algorithm can be used to speed up the eigenvector calculations [191, 238]. Furthermore, Hendrickson and Leland propose a strategy using multiple eigenvectors to compute multiple partitions at once, speeding up the process even more [144]. Nevertheless, RSB on its own is not suited for dynamic partitioning due to its expensiveness [293]. It is difficult to parallelise efficiently [17, 291] and does not maintain geometric locality [316]. Lastly, in its base formulation RSB is not incremental. Van Driessche and Roose however give a formulation of RSB that possesses this property [309].

2.4.2 Local Graph-Based Methods

Given an already existing partitioning, a local graph-based method works with a small overlapping set of partitions. In this work, each partition is assigned to a processing element. If imbalances occur during the runtime of the application, the local method tries to resolve these imbalances by shifting workload from one processing element to another within the same set of partitions. This is especially suited for applications with frequent local changes to the domain graph that require a dynamic load balancing. Even more so, as the methods are naturally incremental [69, 234]. The sets can be defined by partitions having dependencies with each other, in other words they have connecting edges between nodes assigned to them [161, 327]. Another variant considers the connectivity of the underlying hardware [197]. These sets overlap, which makes it possible for workloads to travel through multiple partitions in several iterations of a local method, until good global balance has been reached. Therefore a local method performs best for small local imbalances that are resolved within a few iterations. Local methods are easy to parallelise [69, 234, 180, 318, 106] and are naturally scalable. Sets to consider may be arbitrarily small, starting from a single partition and its neighbours. Therefore, local methods are independent from the size of the computational domain and the number of partitions, or number of processing elements, that should be considered.

Local methods typically are comprised of two steps. In the first step, a map is computed of how much workload (nodes or objects) must be shifted from overloaded partitions to lesser loaded ones to restore a balanced partitioning. Here two main alternatives can be considered, Diffusion-based methods 2.4.2.1 and Demand-Driven methods 2.4.2.2 These methods are elaborated in their respective sections.

The second stage of a local method then selects objects to transfer according to the computed map. Among the most used methods to determine the nodes to exchange a gain criterion is computed, similar to the gain criteria used in Node-Swapping methods 2.4.1.2. Possibilities are the method of Wheat et. al. [327, 328] for unweighted edges. The method only considers the minimisation of the amount of edge cuts and moves the node with the lowest resulting amount of cut edges. An extension for weighted nodes considers the minimisation of the total edge-cut weight [77]. A further extension introduces a relative gain measure. That means including a measure of the gain of the node about to be transferred from the target partitions' perspective. This aims to reduce the number of collisions, a node transfer that is immediately reversed [318]. Pairwise exchanges between partitions are explored in [136]. The gain criterion can be further enhanced by also adding a measure of the number of transfers necessary into its formulation [77, 277, 336]. Here, a configuration with a lower edge-cut weight could be discarded if the cost of reaching that configuration outweighs the benefits. All these methods try to minimise the surface of each partition as this directly minimises the amount of communication between partitions, given an accurate model of communication by the edges. There are applications, however, where other criteria are more important. For example the aspect ration partitions has a larger influence towards solution time for domain decomposition linear solvers [79, 310].

Local partitioning algorithms may take a long time to converge for large imbalances and large domains, where the performance of global methods is far superior in general. But, local methods are a popular choice as part of a Multilevel method to improve the quality of already pre-partitioned graphs (see section Multilevel methods 2.4.3), running only a few iterations. Nevertheless, when the graphs to partition become massive, for example on current supercomputers, where each processing element represents a partition and the number of objects to assign and balance is again orders of magnitude higher, global methods will reach their limits. In this scenario, local methods will become the only applicable dynamic partitioning methods [281, 92].

2.4.2.1 Diffusion

Diffusion is a natural process describing a substance's desire to distribute evenly in space [37]. Discretising the diffusion equation on a mesh given by the connectivity of the objects of the computational domain or the network of the parallel machine allows to define an iterative process to balance out the workload among all partitions. Diffusion algorithms for dynamic balancing of partitions were first proposed by Cybenko in 1989 [66]. In its base formulation, the method is solved using a first-order finite difference scheme. The stencil for this scheme only uses direct neighbours and is thus compact [81]. Using a higher-order stencil, that is including more distant mesh points from neighbours of neighbours improves the convergence of the method, though also sacrificing the locality of the method [323, 322].

To carry out an iteration of the algorithm, each partition evaluates the workload difference between itself and all its' neighbours and transfers workloads according to this difference. As objects are only transferred between partitions that share neighbouring relations with each other, the balanced partitions are incremental by design.

Another type of diffusion method is so-called Dimensional Exchange [78, 329, 336]. Here, a hypercube network is assumed to represent the connectivity of the partitions. The algorithm then loops over all dimensions of the hypercube and the partitions balance workloads with their respective neighbours in that dimension. After all dimensions have been looped over, the workload is balanced. The algorithm can also be applied to non-hypercube graphs by mapping the hypercube to the underlying graph configuration. This however renders the algorithm non-incremental as loads are also exchanged with partitions without a direct connectivity.

2.4.2.2 Demand-Driven

Another local graph-based method is the Demand-Driven model. The idea behind this method, is that underloaded or overloaded partitions request or send workload to neighbours when they detect an imbalance among them [327, 327]. There exist two versions of the model, a receiver-initiated version, where underloaded partitions request work and a sender-initiated version, where the work transfer is initiated by overloaded partitions conversely. Compared to the diffusion model work transfers do happen asynchronously only if demanded and only among a subset of all neighbouring partitions. Furthermore, the workload sharing is only initiated if a process becomes idle or overloaded. Small imbalances in workload that do not affect performance are not considered. The threshold when a partition is considered underloaded or overloaded are application dependent. In practice the receiver-initiated model has proven to be more successful, because the overhead of balancing load is assumed by lesser loaded partitions and does not impact the already overloaded partitions [329].

Similar to the diffusion methods, the neighbourhood model can either be application specific [328] or dependent on the connectivity of the hardware network [197]. Another variant to define neighbourhood can be found in Özturan et. al. [69, 234], who group sets of partitions into trees, based on the request of workload they make. The workloads then subsequently move only between the partitions within the same tree.

The Demand-Driven method has been successfully employed by the multiphysics framework Alya from the Barcelona Supercomputing Center for the solution of coupled problems, for example incompressible and compressible flows, solid mechanics and particle transport [312, 158]. Alya is one of two Computational Fluid Dynamics Frameworks of the Unified European Applications Benchmark Suite (UEBAS) and is also part of the Accelerator benchmark suite of the Partnership for Advanced Computing in Europe (PRACE) [307]. For its initial partitioning, Alya uses Metis [174] for initial partitioning and the library LeWi which implements a Demand Driven model sharing workloads among cluster nodes in a supercomputer for dynamic repartitioning [119, 120]. The code was successfully deployed on the MareNostrum supercomputer of the Barcelona Supercomputing Center [156, 157].

2.4.3 Multilevel Methods

The main drawback prohibiting the use of the above mentioned global graph-based methods for large graphs for dynamic partitioning are their expensiveness, both in terms of runtime and storage requirement. The later can be alleviated by efficient storage schemes, but using one of the methods on a large graph is simply too expensive to justify its cost compared to a possible time save due to a more efficient partitioning. Multilevel methods conveniently solve this problem by allowing the use of expensive partitioners on a reduced representation of the graph. In this regard, Multilevel partitioning methods share many similarities to multigrid methods used to solve systems of linear equations with a hierarchy of representations of the computational mesh.

A generic Multilevel scheme consists of three phases, a coarsening phase, a partitioning phase and an uncoarsening phase, also called refinement. In the first phase, the graph is gradually reduced to coarser representations of the original graph with fewer nodes and fewer edges. This is achieved by contracting sets of nodes on the finer levels into single nodes on coarser levels. The weights of the contracted edges are summed up, parallel edges from the contracted nodes are summed up as well. This procedure is repeated until the resulting graph is small enough such that one of the global graph partitioning algorithms mentioned above can be applied to generate an initial partitioning. Finally, in the uncoarsening stage, the contraction is gradually reversed, with the possibility adding a local improvement of the partitioning, for example using a diffusion method on each intermediate level. Adding to the similarity of Multilevel methods and multigrid methods, coarsening and refinement can similarly be combined as restriction and prolongation. Gradually contracting nodes until the coarsest level and then reversing until the original graph is rebuild resembles a classical v-cycle. Other cycles have been used too, such as the w-cycle which combines more intermediate coarsening and refinement cycles to put more emphasis on partitioning the coarser levels, as well as the f-cycle, where through nested iteration one builds up a good initial guess from the coarsest level [315, 269].

Multilevel methods promise a few advantages. Firstly, on the coarsest level, the most expensive partitioning methods, netting the highest quality partitioning can be used without much impact on the runtime of the method due to the decrease in size compared with the original problem. Although, it is not guaranteed that a good initial partitioning of a coarse representation translates to an equally good partitioning when refined. Still, multilevel methods allow several runs of different partitioning kernels and picking the best outcome while being less expensive than a comparable global method [37]. Secondly, small movements of nodes on the coarsest level translates to large movements of nodes on the finer levels. A good partitioning might be able to be computed faster as less movement is needed for a node to reach its final position. And lastly, the improvements employed during the refinement are expected to perform well, since they start from an already good initial partitioning from the coarser levels.

Multilevel methods were first developed by Bui and Jones for the computation of fill-reducing matrix reorderings [36]. Independently, Hendrickson and Leland also proposed the method for Finite Element mesh partitioning [145] and Hauck and Borriello applied it directly to graph partitioning calling it Optimised KLFM, because they applied Kernigham-Lin / Fiduccia-Mattheyses type Node-Swapping techniques [139]. In fact, Multilevel methods have been established as the de-facto standard for graph partitioning and many authors have published their version of it for various applications [130, 179, 219, 317, 57, 118, 134].

Coarsening
As mentioned, each Multilevel Method consists of three stages, the first being a coarsening in which the original graph is gradually reduced. Karypis et. al. give a comprehensive

overview over different techniques to contract nodes [178, 279]. The simplest being randomly selecting nodes that have connections between them to contract [318]. Heavy-Edge Matching combines nodes with the heaviest edges in descending order. The idea behind is, since the heaviest edges are contracted within single nodes, the coarse graph naturally ends up with the edges with the lowest weights. Partitioning this coarse graph leads to the smallest edge cut weight. Heavy-Edge Matching completely neglects node weights though, which may lead to large imbalances in partition weight. A solution to this problem is to introduce an edge rating function that takes not only the edge weights into account but also the balance between contracted nodes [2, 153, 269, 233]. Edge rating functions have also recently been enhanced using algebraic distance measures gotten from eigenvector analysis [264].

The disadvantage of heuristic methods like Random Matching, Heavy-Edge Matching and Edge-Rated Matching is, even though it performs well in practice and are comparably fast, they can not give any explicit guarantees of the quality of the coarse graph for partitioning. Conversely, there are also a variety of slower methods that give those guarantees. Examples are Greedy algorithms [254] and Path Growing Algorithms [87] as well as their combination [207, 271], which guarantee at least half of the optimal edge weight. Improvements of these are even able to achieve a two thirds guarantee at the cost of a higher complexity [247, 314, 88].

Matching-based coarsening performs well for graphs from applications found in scientific computing. Especially if the emerging partitioners are to be used to assign workloads to processing elements in a parallel machine. There are however more strategies for more irregular graphs and for applications where the quality of the partitioning is more important than a possible very high runtime. These include contraction of multiple nodes at once or coarsening using weighted aggregation. The interested reader is referred to Buluç [37] for a comprehensive overview over these methods and their respective literature.

Partitioning
After the graph has been coarsened, even the most expensive partitioners can be applied without much impact on the overall runtime as the coarsened graph is comparatively small. As such, a large portion of multilevel schemes use some form of Recursive Spectral Bisection as it produces the most optimal partitions [146, 145, 252, 253, 134, 18, 178]. Nevertheless, Karypis and Kumar state that, given good coarsening and refinement strategies, the quality of the coarse level partitioning has only minimal impact [176]. As such they propose a very simple but fast Node-Swapping technique for the initial partitioning and report as good if not better results for various applications [179]. As graph partitioners are usually not incremental and are not suited well as dynamic partitioners, the use of geometric partitioners has also been proposed in this stage to introduce this property into the method [181].

Refinement
After the initial partitioning, the coarsening must be reversed gradually. In addition, during every intermediate step the partitioning is improved by applying some steps of a improvement-based partitioning technique. The main strategy used here is some variant of Node-Swapping that uses a local search to find suitable nodes to exchange 2.4.1.2.

Node-Swapping comes with a very advantageous benefit. In section 2.4.1.2 variants of the technique were discussed that try to move more than a single node or exchange a pair of nodes from one partition with another, but finding suitable sets to transfer is very expensive [8, 80, 135]. If classical Node-Swapping is applied in the refinement stage of a Multilevel method, finding sets is essentially free as they are represented by the collapsed nodes. An alternative is to use a diffusion-based strategy as discussed in section 2.4.2.1 or another local optimisation technique [278]. This is mainly done when the scheme shall be used as a dynamic partitioner to make the method incremental [155, 277, 320, 318].

Multilevel methods are the de-facto standard for graph partitioning. They provide the best combination of partition quality and speed. They are faster than other global methods and provide at least equal quality partitions. Local graph-based methods are highly dependent on the structure of the graph and may need a large number of iterations to match or surpass the quality of the partitioning provided a Multilevel method. As such, there exist a large number of high quality libraries implementing Multilevel Techniques. Examples are Chaco [143], Metis [174], Scotch [243], Party [255] and KaHIP [270]. These libraries have seen great success as static partitioners. However, since they implement sequential versions of the multilevel approach, they are not suited as dynamic partitioners.

Parallel implementations have proven to be rather challenging. As was mentioned, graph-based methods in general are difficult to parallelise, even more so when using these as components of a Multilevel method. Still, there have been efforts to implement parallel Multilevel methods to be used as dynamic partitioners. The most successful are ParMetis[181, 177, 277, 180] and PT-Scotch [242], both parallel versions of their sequential counterparts, as well as Jostle [320, 318, 319].

2.4.4 Hypergraph Partitioning

When using classical graphs to model the decomposition of a computational domain the edges between the nodes model the communication volume necessary to satisfy dependencies between the partitions the nodes represent. It is sensible to work towards a minimal accumulated edge-cut weight when assigning nodes to a partition. In other words, the goal is to minimise the communication volume between partitions. For most applications from the field of mesh-based simulation, modelling the communication volume using a classical graph is a good approximation, which, given the use of good partitioners, leads to good results. But for highly connected, heterogeneous and asymmetric problems as they occur in VLSI design for example [43], the classical graph definition reaches its limits. Here, Hypergraphs can be used to model these problems. Nevertheless, they are also able to more accurately model communication volume for mesh-based problems and consequently a hypergraph partitioning is able to provide even better partitions than regular graph partitioners [49].

In a hypergraph, the nodes similarly represent objects to be partitioned, but in contrast to a regular edge, which models dependencies between two objects, hyperedges represent dependencies between an arbitrary number of objects [48, 49, 53]. This allows hypergraphs to model more complex relationships between objects and allows for a more accurate model of the communication volume for example. Similar to normal graph partitioning, hypergraph partitioning also tries to minimise (or maximise) an objective function, usually the number of cut hyperedges or the accumulated hyperedge-cut weight, while satisfying a balance constraint.

As mentioned before, one of the points of criticism that limits the use of global graph-based methods as dynamic partitioners are possibly high redistribution costs. In a hypergraph formulation, it is possible to model the total communication volume induced by the transfer of messages to satisfy dependencies and in addition, the costs induced by data redistribution. An accurate model of communication and migration costs leads to the highest quality partitions of all partitioning methods introduced. Consequently, hypergraph methods are widely used as static partitioners, for example in the libraries PaToH [48, 49], Mondrian [311] and hMetis [175].

The high quality partitions produced by a hypergraph formulation come with the cost of a more complex model and more expensive algorithms to evaluate good partitions, compared to regular graph partitioning. As such, competitive hypergraph partitioners also use the multilevel paradigm to their advantage [64, 175]. Furthermore, similar efforts have been put into the parallelisation of these methods [50, 303, 304]. Increasing the speed of the method and since the hypergraph model can explicitly model incrementality in its formulation it has been successfully employed as a dynamic partitioner [51, 52]. Parallel multilevel hypergraph partitioners can also be found in Zoltan [30, 74, 29] and Parkway [304].

2.5 Summary

This chapter has given a comprehensive overview over the various partitioning methods currently employed. The two main classes of partitioners are geometric and graph-based methods. Among the geometric methods, different recursive bisections techniques and Space-Filling Curves have been introduced. As these methods are among the global methods with complete knowledge of the complete simulation space, the primitives can be optimally distributed among the participating processing elements. Furthermore, geometric methods are conceptually simple and usually quite fast. Most geometric methods are incremental – only Recursive Inertial Bisection is not incremental – the repartitioning costs are comparatively low. Small changes in the domain only lead to small changes in the partitioning, where most primitives do not need to be moved. These characteristics and their ease in implementation makes them a frequent choice for dynamic simulation scenarios with frequent repartitioning. The main drawback of geometric methods is their lack of dependency information between primitives, for example communication costs. As such, a geometric method cannot incorporate this information into its formulation and dependencies independent from the geometric location of primitives cannot be taken into account.

The other main class of partitioners that were introduced are graph-based methods. In contrast, these methods explicitly model the dependencies between primitives but neglect their location. The graph partitioning problem is a constrained optimisation problem that is much more involved than computing a simple geometric partitioning. Although, optimal solutions exist, their complexity is too high to be computed frequently. In practice, proven heuristics that provide reasonably good approximations are used. Graph-based methods can be further classified into local and global methods, where global methods are able to compute an optimal result of the partition size. Among them are Graph Growing, Node-Swapping and Recursive Spectral Bisection. All of them compute very high quality partitions, however they are very expensive, not incremental and difficult to parallelise efficiently. This makes them unsuitable for dynamic partitioning.

Local graph-based methods on the other hand only have a limited view of the domain graph and try to incrementally improve already existing partitions. Here, diffusion and demand-driven methods were discussed. These methods may take a long time to converge to an acceptable solution, but are independent from the size of the system and the computational domain. Furthermore, these methods are incremental. In practice, they are a popular choice as part of Multilevel schemes to improve the quality of intermediate coarser representations of the domain graph with only a few iterations.

Multilevel schemes are the current state-of-the-art when it comes to graph-partitioning. They try to alleviate the expensiveness of global graph-based methods by applying them on a reduced representation of the graph. The graph is reduced by contracting sets of nodes on the finer levels into single nodes on coarser levels. After computing a partitioning on the reduced graph, the contraction is reversed and the original graph is recovered. The characteristics of the used partitioning method concerning the quality and incrementality are retained from the methods used within the scheme.

For the sake of completeness, the partitioning of hypergraphs was briefly discussed.The insights from regular graph-based methods are also true here. The advantage of hypergraphs is that the model can be augmented by further dependency information, which in turn can be considered in the partitioning. As such, the hypergraph model is more accurate and leads to even higher quality partitions. However, the cost of evaluating these partitions is also increased.

Chapter 3

A Decentral Framework for Numerical Simulation

The advantage of local partitioning methods is their complete independence from globally shared information. As such, their use is not prevented by the size of the machine they should be deployed to. In order to show the viability of this approach, a simulation framework for the numerical simulation of physical systems represented by mathematical models has been developed to incorporate one of these methods. Specifically, a local diffusion-based repartitioning is used. The baseline for this framework is the work of Frisch, who developed a simulation code for fluid problems tailored to massively parallel machines [113]. More accurately, his ideas of a distributed space-tree data structure, where every node of the tree is discretised, not just the leaf nodes, is used. Furthermore, most of the solution procedures using this data structure are used in the new framework as well.

During the development of the new framework, it became obvious that the need to synchronise global information also poses a bottleneck in other parts of the simulation pipeline, in addition to the partitioning phase. Therefore, the basic premise of the new framework has been to refrain from any kind of globally shared information. More specifically, the data structure is completely decentralised with no overarching metadata available to one or more processing elements. This marks a clear deviation from Frisch' concept, who used a central management instance, which kept track of the distribution of the data structure. Furthermore, the new framework tries to limit the use of global communication whenever possible. However, there are parts where global synchronisation can not be avoided. An iterative solution procedure must broadcast a stop signal when a global convergence threshold has been reached, for example.

In this chapter, this new framework is detailed. First, the data structure is introduced, starting with the primitives that make up the structure. The primitives are generated using a hierarchical domain decomposition. The decomposition is comparable to the ones used to define a space-filling curve. As mentioned above an important detail of the data structure is that intermediate states of the decomposition are not discarded. Therefore, the resulting structure has hierarchically overlapping primitives, whereas the decomposition is non-overlapping in space. Again, this data structure is completely distributed in the new framework. That means, all primitives are exclusively partitioned among all processes. Even the overarching macrostructure, made up from the hierarchical primitives on coarser tree levels, is distributed and not duplicated. This renders the complete structure decentral and no part of the structure must be synchronised.

The neighbourhood model is introduced next. It details the relationship between the primitives and between the processing elements, which hold these primitives. Due to the hierarchical structure, primitives have two types of neighbours, spatial and hierarchical neighbours. To conclude the core concepts of the framework, the communication patterns are detailed. There are two main communication models. The data exchange between the primitives, which is mainly used to exchange bordering unknown quantities during the solution process. And second, the data exchange between processing elements, which is used in various management tasks.

In the following, additional modules of the framework that are affected by the decentralisation are introduced. First, the domain generation is illustrated. Next, the adaptive mesh refinement and coarsening (AMR) is detailed. The AMR functionality causes changes to the domain structure during runtime and consequently changes in workload and connectivity of partitions. Therefore, using AMR is the reason the domain needs the dynamic partitioning functionality in the first place. In this context, the metadata updates of the neighbourhood relations caused by AMR are addressed as well. Furthermore, the framework supports additional balance constraints on the domain decomposition. This might cause refinements to cascade through the structure or refinements to be rejected. Both the initial generation of the domain as well as adaptive changes to it from AMR need to adhere to these constraints. Within the decentral structure, these operations are not trivial and are addressed.

For completeness, the solution methods to solve systems of linear equations are described next. These methods are not affected by the decentralisation of the data structure and are taken mostly from the aforementioned work of Frisch. An addition to the new framework is an asynchronous parallel Jacobi method, with a possible second application as a more accurate load estimator.

Finally, I/O, interactive data exploration and computational steering are addressed with an emphasis on adaptions needed to limit global synchronisation.

For parallelisation, the framework follows a distributed memory concept using the Message Passing Interface (MPI), a portable message-passing library standard for C and Fortran [225]. Up to this point, the term processing element was used to describe the wide variety of different manifestations of elements within a microprocessor. For example, a processing element can be a CPU, a GPU or a vector unit. Without Simultaneous Multithreading (SMT) each processing element can run exactly one sequential process. Furthermore, each processing element has exclusive access to a block of memory. In MPI terminology, each process is assigned a unique number, called rank. Whenever the text refers to a rank or a process, it refers to the program running on a single processing element, with exclusive access to its memory block.

3.1 Data Structure

The data structure consists of the grid primitives constructed at the beginning of the simulation using a hierarchical domain decomposition approach. Both the primitives and the domain decomposition are explained in more detail hereinafter. One uncommon feature of the present framework is that it also keeps intermediate coarser representations generated during the hierarchical decomposition. As such, the primitives that make up the data structure are overlapping in a hierarchical sense but not in a spatial sense. Primitives that discretise the same domain are connected and therefore extend the neighbourhood relations between primitives and of the processes they are assigned to in hierarchical direction. This extension aims at improving the convergence of local partitioning methods. Furthermore, the coarser representations are used in parts of the framework unrelated to the partitioning, namely the solution and visualisation modules (see Section 3.6)

3.1.1 Grid Primitives

To simulate a physical phenomena described by a continuous mathematical formulation, the first step is to transfer the continuous model into a discrete representation that can be evaluated by a computer. That means, the values of continuous functions must be transferred to discrete points in the domain. These points form a grid at which the known and unknown quantities are defined. The present implementation uses regular grids, meaning the domain is segmented completely in axis-parallel, right-angled regions with equidistant length along each axis. Each grid point can be indexed by an unique triple (i, j, k) in three dimensions, representing one location in the continuous domain at location (x_i, y_j, z_k).

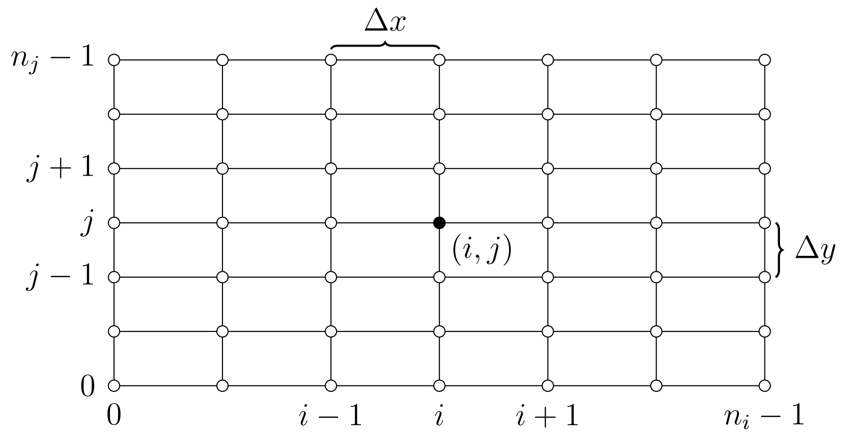


Figure 3.1: Two dimensional regular finite difference grid.

A generic two dimensional grid for a finite difference discretisation is shown in Figure 3.1. In total, the continuous domain is discretised by $n_i \times n_j$ grid points. Neighbouring grid points in each cardinal direction can be reached by increasing (east and north) or decreasing (west and south) the respective index i in x-direction and j in y-direction. As the spacing of the grid points is equidistant, Δx_i is identical for all i. The same is true for

Δy_i. Therefore, one can easily compute the location of the point with indices (i, j) in $\mathbb{R}$, $(x_i, x_j) = (x_0 + i \cdot \Delta x, y_0 + j \cdot \Delta y)$. In three dimensions, an additional index k is added which behaves identical to the former two. Top and bottom neighbours are identified by increasing or decreasing the index. The location z_i in $\mathbb{R}$ corresponds to $z_i = z_0 + k \cdot \Delta z$.

In the finite difference method, ordinary or partial differential equations are transformed into a system of linear equations. The derivatives are approximated by an algebraic expression, the finite difference, defined at the grid points. The number of grid points used to approximate the derivatives determines the order of the approximation [129]. If values are needed at locations other than directly at the defined grid points, they are interpolated. The most common way to approximate the values at these locations is bilinear interpolation in two dimensions and trilinear interpolation in three dimensions respectively. That means, that the values at directly surrounding grid points are used and their contribution relative to their distance is added to find the value at the location. Higher order interpolation is also possible, incorporating more and further away points and can increase the accuracy of the interpolation.

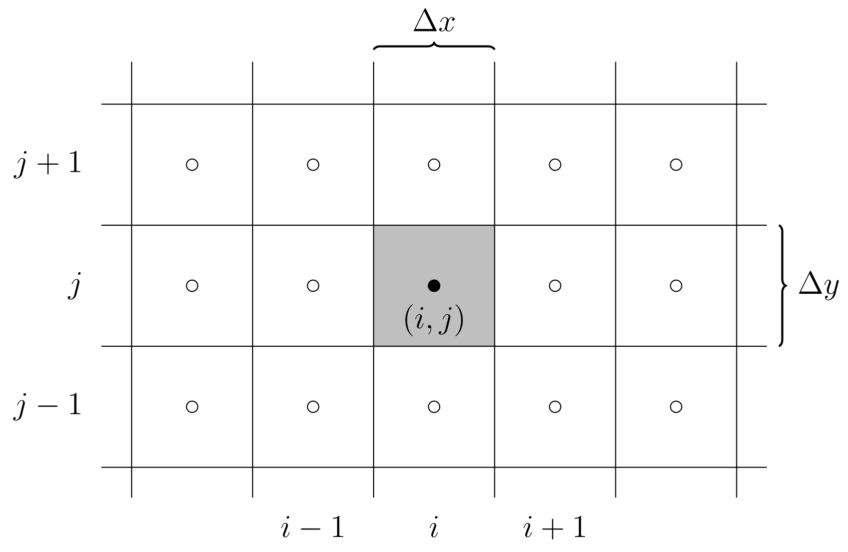


Figure 3.2: Two dimensional cartesian finite volume grid

The data structure used here is not exclusive to finite difference methods, but can be applied to other mesh-based methods as well. In Figure 3.2 an exemplary finite volume grid is shown. Here, the grid point at index location (i, j) represents the complete control volume, also called cell, surrounding it. In other words, the value stored at the grid point represents the integral over the cell volume. As the spacing of grid points along each axis is equidistant, the distance between the grid points Δx and Δy correspond to length and width of the cells. The index location (i, j) can also be understood at the center of the control volume. This allows to easily compute the extent of the cell. For example, the left lower corner (south-west) is at the location $(x_0 + i \cdot \Delta x - \frac{1}{2}\Delta x, y_0 + j \cdot \Delta y - \frac{1}{2}\Delta y)$. The grid in the Figure is a special case of regular grids, where the spacing along each axis is equal, i.e. $\Delta x = \Delta y$. In other words the control volumes are squares (2D) or cubes (3D). In finite volume computations the value at cell faces (edges in 2D) is often needed. The default way to approximate the values is to simply linearly interpolate between the two cells sharing the face. This grid primitive is used in the present implementation. Using the mid-point rule to approximate the volume and surface integrals of the control volume, a simple stencil-based finite difference scheme can be used to update the cell quantities (cf. [113]).

3.1.2 Domain Decomposition

As mentioned above, we follow the idea of space-trees to decompose the simulation domain. In the beginning, the computational domain is represented by a single node that encloses the complete spatial domain. This is the root node of the tree structure at depth $d = 0$. Now, the tree is successively build by refining the nodes on the current deepest level. To refine a node, the domain is subdivided along its cardinal directions. A bisection in each direction results in four child nodes for every refined node in two dimensions and eight child nodes in three. Accordingly, trees using a bisection are called quadtrees (2D) or octrees (3D).

After the tree has been generated, each leaf node is commonly discretised using an individual grid. As such, the discretised computational domain is already decomposed into primitives ready to be distributed to the MPI ranks. Space-filling curves are especially well suited to distribute these grids. A traversal through the tree gives a neighbourhood preserving order of grids that can simply be cut into equal sized chunks, according to the number of participating processes. To be able to apply an SFC partitioning, the complete space-tree including the overarching macrostructure, which consists of all refined nodes, has to be available. Storing space-trees uses some very efficient storage schemes, therefore memory limitations are currently not an issue. Changes to the tree have to be broadcast to all processes though, the effort for which scales with the amount of processing elements used.

Most computational domains are not uniform. That means they exhibit regions of high and low interest. These regions are defined either by the requirement of the numerical scheme or by the specific design purpose of a numerical simulation. A high numerical accuracy is required for example in fluid flow simulations in the vicinity of obstacles in the flow regime. For once, the obstacle must be represented accurately, using a high enough resolution and furthermore, the flow structures influenced by obstacles are often the main interest behind a simulation. This requires to refine the space-tree until the regime can be accurately captured to resolve all occurring phenomena. A uniform refinement of the space-tree, where every node is subdivided until the discretisation of the leaf nodes result in the desired resolution will inevitably result in many wasted resources, as all regions are resolved with the same, possibly very high accuracy. A solution is to adaptively refine the space-tree only towards interesting features in the computational domain. In Figure 3.3, an exemplary adaptive two dimensional decomposition is illustrated.

On the left hand side, the space discretisation through the successive refinements is shown. At the bottom, the grids discretising the leaf nodes of the tree are accumulated. Under the presumption that every grid consists of an equal number of grid points, regardless of the space it discretises, the resolution is increased towards a region of interest at the

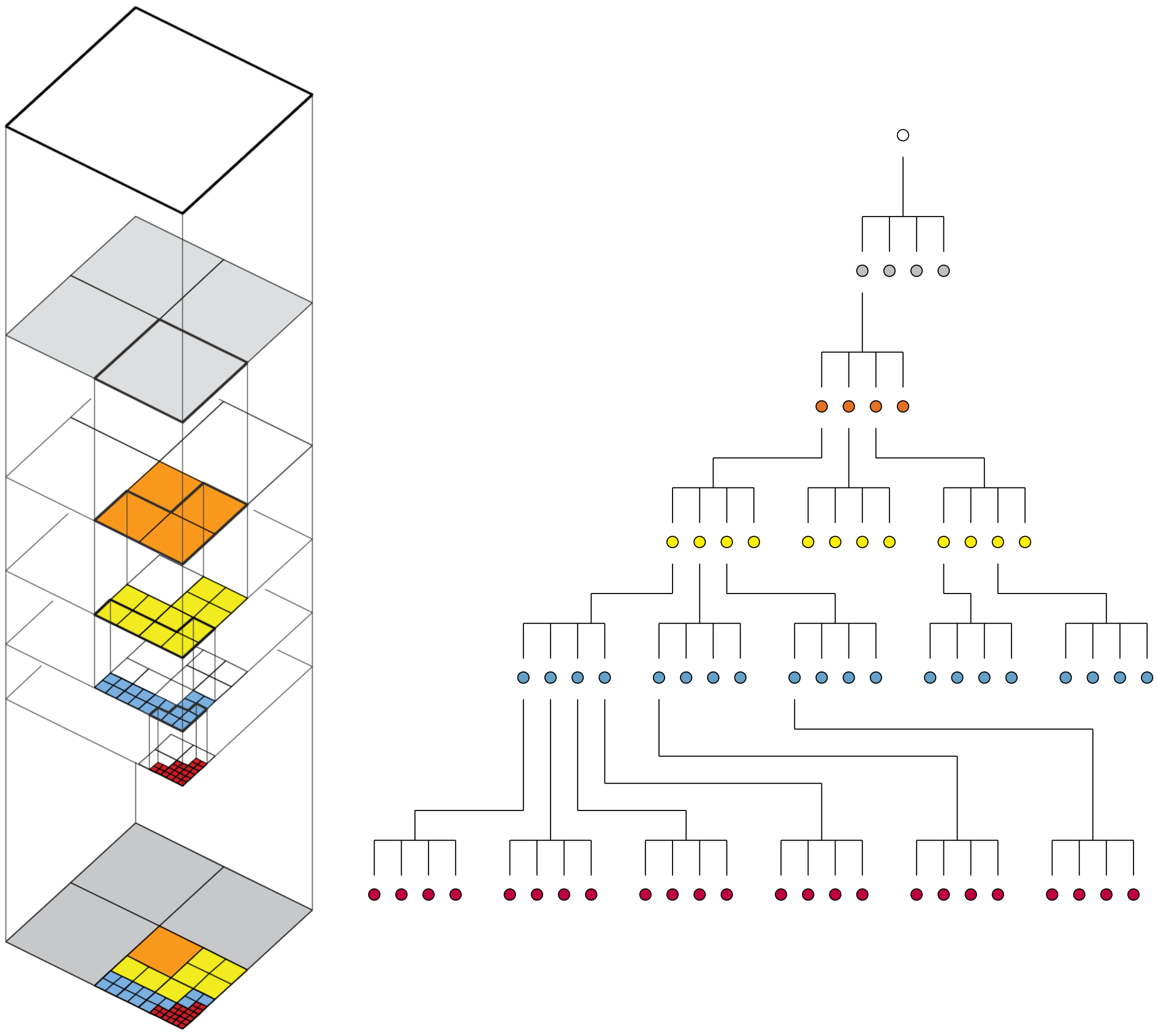


Figure 3.3: Spatial and hierarchical adaptive domain decomposition using a quadtree in two dimensions. The underlying tree is illustrated on the right, the spatial decomposition is shown on the left. Grids discretising the leaf nodes are accumulated in the bottom left. The left illustration is taken from [245].

bottom corner of the domain. On the right hand side the corresponding space-tree is illustrated. As every node refinement results in four child nodes, two in each cardinal direction, it is a quadtree. Using a quadtree, or an octree in three dimensions, is too restrictive for complex domains, though. For example, when simulating channel flows with one elongated direction, a bisection would generate degenerated grids with large aspect ratios, which can cause numerical errors. As such, the present framework allows varying subdivisions between one and eight in each cardinal direction, resulting in up to 49 child nodes in two dimensions and up to 343 child nodes in three dimensions. Restricting the subdivision to eight is caused by the grid identification scheme detailed in section 3.1.4, though numerical issues prohibit a recursive subdivision of more than five anyway [113]. The parallel adaptive generation of the tree with respect to refining towards geometric features is detailed in 3.4.

The data structure is decomposed into individual grids given by the space-tree, the discretised derivatives are defined across multiple data points however. Therefore, grids need to access values at grid points held by neighbouring grids to compute the derivatives at their boundary. For fast retrieval, each grid is surrounded by a so-called ghost cell halo, additional mesh points, to store values from neighbouring grids at their respective side. The thickness of the necessary ghost cell halo is determined by the approximation of the derivatives. Higher-order derivatives or more accurate approximations need more neighbours and warrant more ghost cell layers. All grids have full ghost cell halos. This includes the grids located at the domain boundary. This serves to have a uniform grid description and the ghost layers which would theoretically represent values outside the domain may be used to impose boundary conditions.

Figure 3.4 illustrates the decomposition for a uniform mesh configuration. A single 4×2 finite volume grid is decomposed into two 2×2 grids. The grids are coupled using a single layer of overlapping ghost cells (illustrated in gray) at their respective boundaries. After every iteration of a solution method, the values in the ghost cells are updated using recently computed values of the respective neighbour. During the computation, the values in the ghost cells are treated as boundary conditions and are not updated. Therefore, the solution of such coupled grids can be separated into two phases: a computation phase where a grid's own cells are updated and a communication phase in which ghost layers are updated following the arrows in the illustration. These types of domain decomposition methods for the solution of boundary value problems are commonly known as Schwarz Methods [290] after Schwarz, who was the first to prove convergence of the method for the Laplace equation.

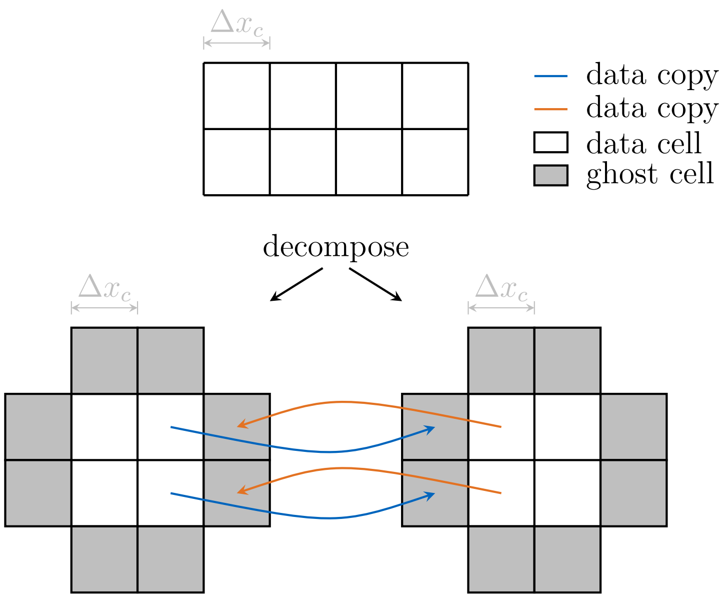


Figure 3.4: Two dimensional cartesian finite volume grid

An exemplary two dimensional case of a decomposition of a non-uniform domain is illustrated in Figure 3.5. Here, the decomposition produces two uniform grids. A 2×2 and a 4×4 finite volume grid. As the decomposition is generated by discretising the space-tree nodes, the accumulation of all grids who discretise leaf nodes may produce a non-uniform configuration. Each individual grid will always be uniform though. These grids are again extended with ghost cells on all sides. The configuration of the ghost cells is conceived as if the neighbouring grid would be of equal spatial size as the actual grid. This, in addition with having uniform grids, allows to use the same stencil, describing the discretised partial differential equations, across all grids, regardless of their size.

Updating of ghost cells' values becomes more involved though. As the overlapping halos have different sizes, values cannot simply be copied anymore. Updating coarser ghost cells representing data on finer grids involves an aggregation. All data values from cells that are covered by the ghost cells must be aggregated according to the amount of overlap and the values have to be interpolated to the same position where the data on the coarse ghost cell is located, that is its cell center. Fortunately, as the discretisation of grids is self-similar, meaning finer grids are simply subsections of coarser grids, a coarse cell always covers a number of fine cells exactly. As such, the aggregation degenerates to a simple averaging of all covered cells. In the Figure, this is illustrated by four cells averaging their values and updating the corresponding ghost cells marked by orange arrows. To be able to use the same stencil, the data in the finer grids' ghost cells must be correctly prolongated. The location of the cell center, where the data values are stored is different between the coarser grid and its finer ghost cell representation. Therefore, the value is linearly interpolated from the coarse grids' cell center to the fine grids' cell center (dashed blue arrows).

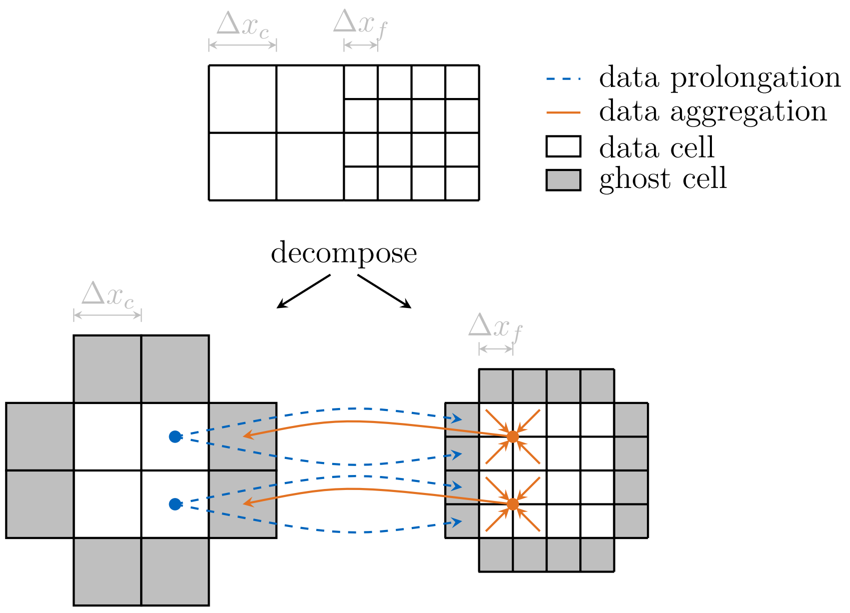


Figure 3.5: Two dimensional cartesian finite volume grid

Linearly interpolating between neighbouring grids of different size introduces an error into the computation. For numerical accuracy, the tree-based decomposition is subject to a strict 2:1 balance constraint. In other words, neighbouring grids are not allowed to differ by more than one refinement level, to restrict the interpolation error between grids. This balance constraint is common in space-tree-based decompositions [164]. However, if the tree is not available in its entirety, as it is the case for the present framework, the balancing becomes more involved. This impacts the initial generation of the tree data structure and alterations to the tree using the adaptive mesh refinement and coarsening features. Establishing and keeping the tree balance is covered in their respective sections 3.4 for the initial generation and 3.5 for AMR.

As mentioned above, the present framework discretises all nodes of the space-tree, even refined ones. The data structure therefore represents the simulation domain in various resolutions throughout the whole space-tree hierarchy. All grids are exclusively distributed, in other words there is no globally shared macrostructure that connects all partitions, which makes the data structure completely decentralised. This has implications for the neighbourhood model 3.2, as now, hierarchical neighbours have to be considered, too. Furthermore, the coarser grids are used in a custom-tailored multigrid method for solving the attached mathematical model 3.6.2 and in an online visualisation scheme, which allows an overview of large-scale simulation data.

3.1.3 Boundary Conditions

Boundary conditions are necessary constraints to obtain the solution of a boundary value problem. Common conditions for a unknown function are Dirichlet and Neumann boundary conditions. The former prescribe the value of the function itself at a specific point in the domain, the later prescribe the value of the derivative in normal direction at the boundary of the domain. Imposing boundary conditions for the present grid primitives is simple. Each cell stores an additional value describing an imposed constraint. Here, the unphysical ghost cell halos at the boundary of the domain have a practical application. They can be used to impose various problem specific boundary conditions. For example, in a fluid calculation, a wall with a slip condition may be represented by imposing a Dirichlet condition on the normal component of velocity in the fluid, as the wall is impenetrable and a homogeneous Neumann condition (i.e. a zero derivative) of the tangential velocity component. The later can be achieved by requiring the velocity component to be identical between the fluid cell and the halo cell, which represents the wall.

Boundary conditions are also used to turn off specific sections of the simulation domain. Wherever there is a Dirichlet condition imposed, the value of a variable in the cell is prescribed, rendering a computation unnecessary. This allows for example, to mark solid objects in a fluid domain or embed a more complex, non-rectangular simulation domain within the rectangular grid description. The main drawback of this method is that boundary conditions and geometric features can only be represented by full cells. That means, normal directions for Neumann conditions are always aligned to the grid axes and object boundaries follow a stepwise approximation. An adaptive refinement of the simulation domain near those structures may alleviate these effects at the cost of a higher computational effort and numerical errors introduced by a higher unphysical surface roughness.

The idea to embed a complex domain into a simpler fictitious domain, which can be easily discretised with a structured mesh has been used for example in the finite cell method (FCM) for solid mechanics problems [239, 89]. Here, the fictitious domain approach is combined with high-order finite elements to approximate the solution fields. The geometry is represented by using an adaptive integration scheme, which independently from the finite cell grid refines towards geometric boundaries using a hierarchical domain decomposition and places quadrature points in the refined sub-cells. Combining the h- and p-versions of the finite element method, FCM achieves optimal rates of convergence, when the mesh is refined, and exponential rates of convergence, when the polynomial degree is increased (cf. [276]). The method has since been adapted for fluids, too [337]. Similar methods have also been developed for finite volume and finite difference methods with a focus on fluid mechanical problems. Here, these methods are called immersed boundary methods and work has been done by Fadlun et. al. [96], Kim et. al. [185] and Tseng and Ferzinger [306] for example. However, as this was not the main focus of the present work, these methods have not been implemented yet.

3.1.4 Grid Identification

In a completely decentralised system, each grid needs to be uniquely identified not only within a specific process but also globally. For this purpose, a 64 bit wide long integer was chosen, again inspired by the work of Frisch [113]. In Frisch' implementation the first 32 bits of the integer are used to describe the rank in an MPI-based implementation the grid is assigned to. The second 32 bits, called tag, are split into a grid identifier, GID, unique to the process the grid is assigned to and a hash. The hash itself consists of nine bits, of which always three are used to encode the position along one direction of the grid in the local coordinate system of its parent grid. Using three bits per direction gives $2^3 = 8$ possible index locations per direction, from where the subdivision limit results.

The original implementation used 22 bits for the encoding of the GID, which enabled the assignment of a little over four million grids to each rank. However, no simulation has been reported that used more than a fraction of the maximum possible grids per rank. The present framework uses 17 bits to encode the local grid identifier which still allows to assign a little over 130.000 grids to each process. As rank and GID are sufficient to uniquely identify a given grid across all processes, the left over bits in the UID can be used otherwise. The encoding the framework uses is illustrated in Figure 3.6.

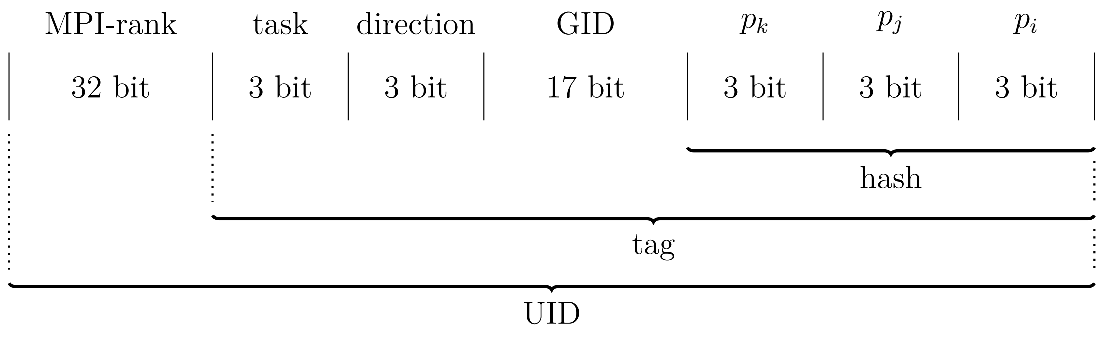


Figure 3.6: Encoding of the global unique identifier (UID) of a grid

In addition to rank, GID and hash, three bits each encode a task and a direction. That allows to encode a wide variety of tasks directly into an MPI message header without any actual data transfer, as all information is stored in the message's tag. For example, updating neighbourhood metadata after the deletion of a grid is one of the eight possible tasks encoded. The GID identifies the grid which needs to update its metadata on a specific process, the direction is able to refer to a specific neighbour grid (there are six geometric neighbours, one parent and a variable number of children). In case the deletion refers to a child, the hash is used to identify the child uniquely. The task encoding can also be adjusted depending on the context of the message, for example during metadata updates following AMR operations there are different tasks as updates after grid migrations. Therefore, the encoding represents an efficient solution to many organisational tasks in the framework. This is also further detailed in their respective sections.

3.2 Neighbourhood Model

Each grid has two types of neighbours, hierarchic neighbours and spatial neighbours. Hierarchic neighbours are grids generated when discretising parent or child nodes from the space-tree refinement. They all represent the same domain or parts thereof in different resolutions. Spatial neighbours are the “real” neighbours, located adjacent to a grid in the simulation domain. The framework’s neighbourhood model only considers spatial neighbours between grids on the same refinement depth of the tree. These neighbours discretise the domain with the same resolution. Neighbours are further limited to the ones which share an edge in two dimensions and a face in three dimensions. Figure 3.7 illustrates all hierarchic and spatial neighbours of a node (orange checkers) in two dimensions. A node has exactly one parent (gray on top level). The number of child nodes depends on the subdivision chosen. Here, a bisection in each cardinal direction per refinement step was used, which amounts to four children (yellow on bottom level) per refined node. In two dimensions, a grid has at most four spatial neighbours on the same refinement level (solid orange). These neighbours share a common edge. In three dimension, a grid has at most six spatial neighbours, with which it shares a face. Nodes at the domain boundary have fewer neighbours. If the domain is non-uniformly refined, nodes may also have fewer neighbours as spatial neighbours are only considered on the same refinement level.

There are a number of extensions conceivable to the neighbourhood model. A sensible choice is to include also spatial neighbours on different refinement levels. With the present approach, in an adaptive configuration, data exchanges between neighbouring grids on different refinement levels have to be conducted through a common neighbour. Particularly, the spatial neighbour of the coarser grid which is simultaneously the hierarchical parent of the finer grid. There can never be more than one intermediate neighbour because of the 2:1 balance constraint of the space-tree. To include these neighbours in the model does however only make sense for leaf grids in a non-uniform refinement configuration. Next to the added storage requirement, extending the model in this way would add another level of complexity from an implementation perspective. Another possible extension to the model is to include also neighbours at corners in two dimensions and ones that share corners and edges in three dimensions. This model’s advantage is the possibility to use more complex stencils which require also cells on these added neighbours. The tighter coupling again adds a larger storage requirement, more communication to keep ghost cells updated and an increase in algorithmic complexity.

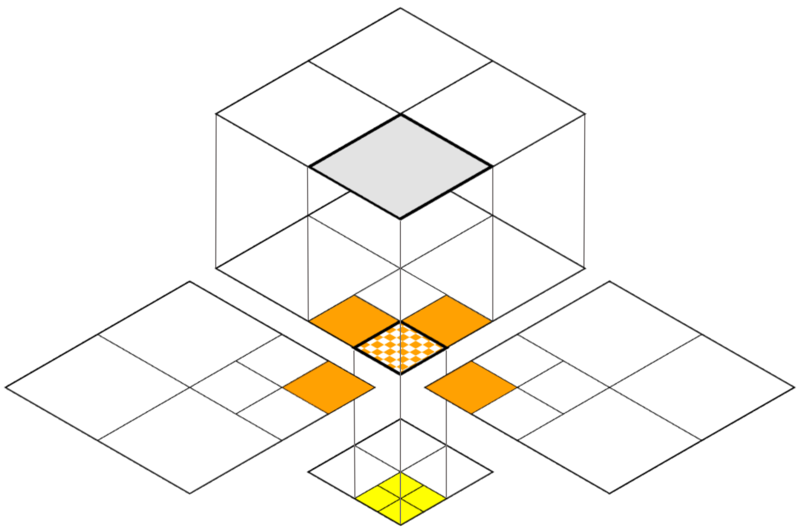


Figure 3.7: Illustration of all geometric and hierarchic neighbours of a grid in two dimensions. When a node is refined, the domain is bisected in each cardinal direction, spawning four child nodes.

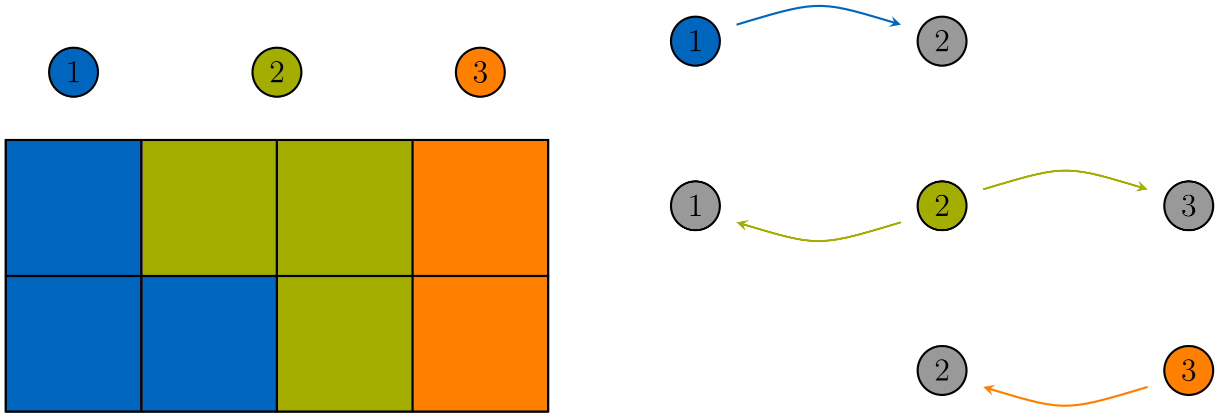


Figure 3.8: The left shows a sample distribution of eight grids to three processes. According to the neighbourhood relations between the grids, the processes holding these processes inherit these neighbourhoods, shown on the right.

After partitioning the domain, each process is assigned a number of grids. Each of those grids has a set of neighbouring grids, which either are assigned to the same or to a remote process. Different processes which hold neighbouring grids are defined as neighbour processes and regular communication must be established among them. Thus, each process has a subset of all processes as neighbouring processes. These subsets are overlapping and unique to a process. Additionally, their union contains all processes, meaning that every process has at least one other neighbour and is reachable. Figure 3.8 illustrates this model. The grids of a sample domain are distributed among three processes. Three grids on the left side of the domain are assigned to process one, three grids in the middle are assigned to process two and the two remaining grids on the right are assigned to process three. The neighbourhood given by the spatial relations between the grids is inherited by the processes which hold them. Grids held by process one are neighbours of grids held by process two. Similarly, processes two and three inherit their neighbourhood from assigned grids with neighbourhood relations. Processes one and three have no neighbouring grids and therefore are not considered neighbouring processes. The process neighbourhood is illustrated on the right side of the figure.

As will be discussed in more detail in the following section, the neighbourhood model directly influences the communication patterns in the framework. Neighbouring processes need to regularly communicate with each other to exchange information. These processes holding neighbouring partitions are also the ones directly communicating with each other in the local improvement method for repartitioning. Therefore, adding second level neighbours extends the number of processes exchanging grids in one iteration of the chosen method, resulting in a higher-order discretisation of the diffusion and a faster convergence towards a good balance. Apart, from the added storage requirement to include second level neighbours, this would render the method less local and therefore less scalable. To find an optimal balance between scalability and extent of the neighbourhood model is not trivial.

As a proof of concept, the framework in its current state implements the above illustrated minimal viable set of neighbours to support all necessary data exchanges during a simulation run. The main ones include exchanging problem variables during the solution process and alterations to the domain graph, which must be communicated to all affected processes. Other tasks are conceivable as well, for example user-generated inquiries such as steering or visualisation tasks. Extending the neighbourhood model in one of the discussed directions and study its implications are viable possibilities for future research though.

3.3 Communication Patterns

As far as possible, each process only communicates with its neighbouring processes to minimise global communication. At the current state, there are only two global communication calls in the I/O module for writing and reading checkpoint files, all other modules follow a completely decentral approach. Each grid has exactly one parent, a set number of children, determined by the given refinement subdivision, and a set number of neighbours. Each process holds a number of grids, with a maximum number determined by the memory available on the hardware. This means that there is an upper bound to the number of grids each process holds, which in turn have a bound number of potential neighbour grids lying on other processes. This upper bound of neighbouring processes is completely independent from the total size of the domain graph, theoretically allowing scalability without any communication bottlenecks. In practice though, the hardware network is not scalable infinitely.

There are two types of communication happening frequently during a simulation run. The first is the data exchange of individual grids during the communication phase of the solver procedure to update ghost cell data in-between stencil iterations. Furthermore, data aggregation and prolongation is used not only to update ghost cells across different refinement levels, but also to synchronise coarser representations and to exchange residuals and errors in a multigrid-like approach (see section 3.6.2 for details). The second is a cycle through all neighbouring processes needed to exchange information and metadata updates for the partitioning and AMR modules. Both are detailed hereinafter.

3.3.1 Grid Data Exchange

As mentioned above, the solution of the underlying system of equation uses iterative stencil loops to smooth towards the solution. These demand updating neighbour values stored in the ghost cell halos after each sweep. The use of 5-point 2D and 7-point 3D stencils requires an exchange of ghost halos along grid edges in two, and along grid faces in three dimensions. Spatial neighbours across grid corners are not needed using these stencils, but if needed, would require an extension of the neighbourhood model. If the domain is uniformly refined, a simple horizontal exchange between grids on the deepest refinement level would suffice to update all ghost cells with the latest values. If the domain is not uniformly refined, some grids do not have direct spatial neighbours. Their ghost cells are updated from a coarser neighbour and vice versa via a common neighbour grid. A direct connection to a neighbour on a different level could also be conceived, the increase in algorithmic cost and complexity of an enhanced neighbourhood model was not deemed worthwhile so far. One could argue that a ghost cell update could be performed in general via the hierarchical neighbours by aggregating the data on successively coarser levels and then updating the cells top-down. This would allow to completely relinquish spatial neighbour connections. Degrading the accuracy of the solution due to aggregating and interpolating the data prohibits this, but might be advisable for non-numerical applications.

To complete all occurring cell data and ghost cell data updates across the different refinement levels, three communication stages can be conceived. In the first stage, termed bottom-up transfer, data is aggregated on children grids and sent to their respective parents. Next, nodes on the same level transfer geometric neighbouring data in a horizontal exchange. Finally, data is transferred from parent nodes to their children, called top-down transfer. In uniform refinement configurations with only simple methods to relax towards the solution, it would be sufficient to only horizontally update the leaf grids, to accelerate the solution however, a custom tailored multigrid-like method is build on top of the data structure. A multigrid method, as the name suggests, requires multiple grids discretising the same domain in different resolutions. Discretising all nodes of the domain tree, resulting in exactly these grids required, is one of the reasons for the duplication of discretised domain. In the multigrid method (see also 3.6.2) residuals are aggregated and send upwards, using the same communication structure as in the bottom-up stage. Vice versa, the top-down structure is used to prolongate error values to correct the solution on finer grids in the multigrid method.

We compute solutions on the finest refinement level, as here the domain is discretised in the highest resolution. Yet, as soon as coarser representations of the solution are required, for example for visualisation or I/O, the solution must be synchronised stepwise bottom-up until the root grid is reached. The three communication stages for grid data exchange are described in more detail in the following. Afterwards, the interaction of the different stages to update ghost cells across different refinement levels is discussed in more details.

3.3.1.1 Bottom-Up Grid Data Transfer

The bottom up transfer is used to send cell data values from finer grids to their parent grid on the next coarser refinement level. Ghost cell values are never updated in this stage. Depending on the subdivision spacing chosen, multiple values are aggregated and send to the parent to update its cell data values. Using a common octree for decomposition in three dimensions for example, one cell on the parent grid discretises the same domain as eight cells on the finer grid. The values of these eight cells are aggregated and the result is used to update the coarser cell's value. In Figure 3.9 the bottom-up scheme is depicted for an example configuration in two-dimensions and a quadtree refinement.

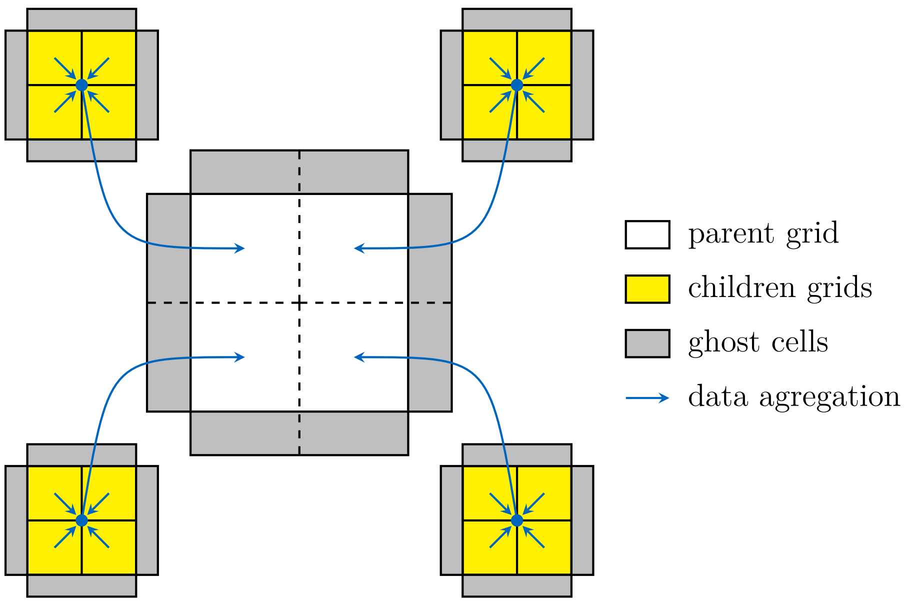


Figure 3.9: Bottom-up communication stage to aggregate and send cell values from the children to their parent grid.

Using an MPI parallelisation strategy, the scheme to communicate through the complete tree structure is implemented as follows. First each non-leaf grid issues a non-blocking `MPI_Irecv` call for every child grid. Next, each process sorts all its assigned grids by their reverse refinement level, meaning grids on the highest level are first, followed by grids on lower levels. This allows to iterate through all grids from most to least refined. Iterating through all grids, leaf grids use a blocking `MPI_Send` to send their data values to their parent grids. Non-leaf grids issue a blocking `MPI_Waitall` call to wait for the arrival of all their children's messages, aggregate their data and use another blocking `MPI_Send` call to transfer the data to their parents in turn. The cascade stops at the root grid, which has no parent anymore. Data travels gradually through the refinement levels and the order in which a parent receives data from its children is irrelevant due to the non-blocking receive calls.

3.3.1.2 Horizontal Grid Data Exchange

In Figure 3.10 the horizontal grid data exchange is illustrated. Grids on the same refinement level, with a direct neighbourhood relation, update ghost cell data with the latest values from their neighbours. Data cells on the sender correspond exactly to their respective ghost cells on the receiver side, therefore no aggregation or projection has to be applied. In contrast to the bottom-up stage, strictly only ghost cells are updated, regular data cells are never touched.

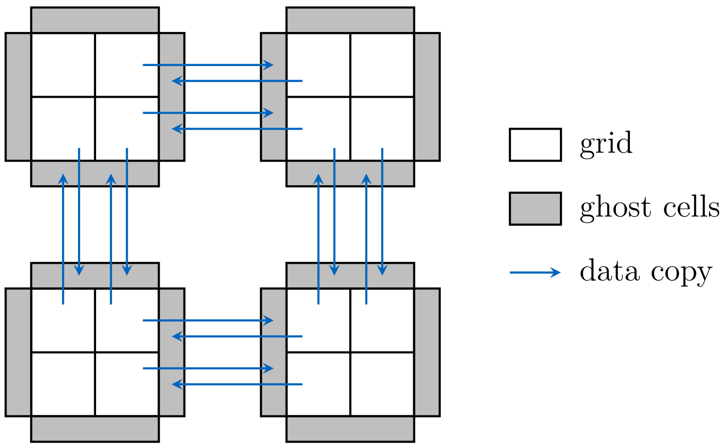


Figure 3.10: Horizontal communication stage, transferring cell values between geometric neighbour grids to update corresponding ghost cells.

Implementation of the horizontal exchange is very simple. Each grid that needs to communicate horizontally issues a non-blocking `MPI_Irecv` for every neighbour it exchanges data with. Afterwards, the grids send their data values using blocking `MPI_Send` calls. After a grid has send all its data, a blocking `MPI_Waitall` stalls the execution until all neighbouring messages have been received.

3.3.1.3 Top-Down Grid Data Transfer

The last stage, the top-down stage, is used to prolongate cell data values and ghost cells from parent grids to their children grids. As mentioned before, data values and especially ghost cell values are usually not representing the exact same domain and are not stored at the same location between finer and coarser grids. Therefore, after receiving data from parent grids, data values are linearly interpolated (bilinear in two dimensions and trilinear in three), taking their fine grid locations into account. Figure 3.11 depicts both the data prolongation to regular cells (blue arrows) and to ghost cells (dashed orange arrows).

The implementation of the top-down data transfer for a complete update of all refinement levels is similar to the bottom-up stage, just mirrored. First, every refined grid issues a non-blocking `MPI_Irecv` call. The sorted list from the bottom-up stage is still valid, therefore, it can again be used to iterate through all grids assigned to a process. This time however, in reverse order from the grids on the lowest to highest refinement level. The root grid starts the cascade sending data to all its children using blocking `MPI_Send` calls. Subsequently the children grids, issue an `MPI_Wait` call, suspending the process

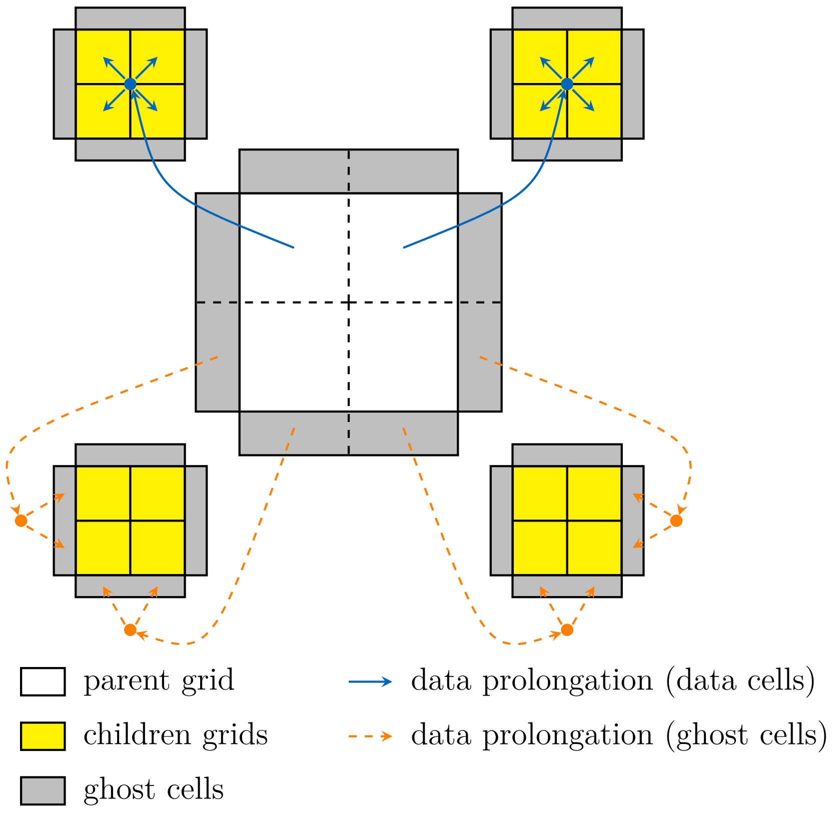

parent grid
children grids
ghost cells
data prolongation (data cells)
data prolongation (ghost cells)

Figure 3.11: Top-down communication stage, transferring cell values from a parent grid to update regular cells and ghost cells. Data is received into buffers projected to their correct locations.

until the receive call is completed. After extracting and interpolating the cell values, a subsequent blocking `MPI_Send` call forwards the values to all children. Grids on the highest refinement level do not forward anything anymore.

3.3.1.4 Ghost Layer Exchange in Uniform and Adaptive Refinement Structures

So far, the implementation aspects for the bottom-up and top-down stages illustrated an information exchange through the complete grid hierarchy. This is needed in the multi-grid solver or when synchronising the solution across all levels for partial visualisation for example. The ghost cell exchange can be limited however, depending on the refinement structure and the application. If the space-tree is completely uniform, all ghost layers, except the ones at the domain boundary, can be updated by pure horizontal communication. Furthermore, only leaf grids have to update their ghost cells, horizontal communication on coarser levels is not necessary.

When the space-tree refinement is non-uniform though, horizontal communication to update all required ghost cells is not sufficient anymore, as some leaf grids do not have spatial neighbours on the same refinement level. In Figure 3.12 the procedure to update those values is illustrated. First, the bottom-up stage is used to update cell values on the parent grid of the fine grid (blue arrows). Next, the horizontal stage exchanges cell values to update the ghost cells between the parent and its spatial neighbour (green dashed and dotted arrows). Finally, the updated ghost cell values can be forwarded from the parent to the child grid using the top-down scheme (orange dashed arrows). Important to note is that the order of operations, bottom-up, horizontal and top-down must happen in this order to ensure a correct ghost cell update.

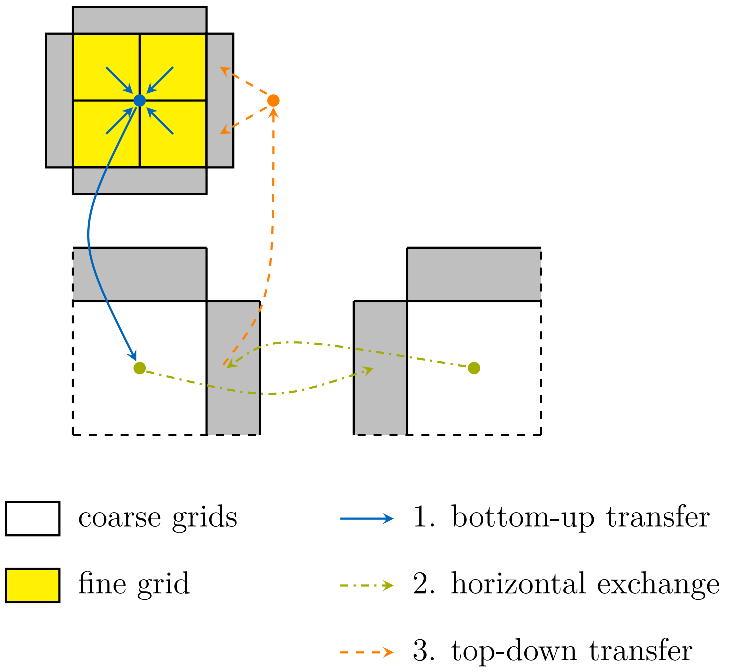


Figure 3.12: Combination of bottom-up, horizontal and top-down communication stages to transfer ghost layer data between neighbours on different refinement level via a common neighbour grid.

In [113], Frisch uses the ghost layer exchange throughout the whole domain structure. This provides a simple update scheme, whose costs have been shown to be reasonable through a performance evaluation. Nevertheless, updates of ghost cells across all grids still is unnecessary. Therefore, the scheme has been improved, only including grids that actually are required to update their ghost cells. All leaf grids have to horizontally communicate. In addition leaf grids with one or more missing spatial neighbour need to exchange data with their parents. Leaf grids easily can determine this for themselves. Finally, non-leaf grids used to forward ghost cell information also need to participate. Figuring out which ones to include requires one horizontal communication between all spatial neighbours in the domain. This communication transfers whether a neighbour is a leaf grid or not. A non-leaf with leaf neighbours must forward ghost cell data.

This is implemented simply using a set of flags for non-leaf grids. Which are set after the domain is generated, and updated if changes to the refinement structure occur. As aforementioned, leaf grids do not need additional flags, if one of their neighbourhood references is empty, they need to update their ghost cells via their parent. Non-leaf grids store whether their spatial neighbours are leafs, allowing them to know with which they need to horizontally exchange ghost cell data.

3.3.2 Process Data Exchange

The second type of communication in the framework is the information exchange between neighbouring partitions, that means between processes that hold sets of grids with neighbourhood relations between each other. A common reason for a process data exchange is for example the exchange of loads between neighbouring process within the dynamic partitioning module.

A complete exchange of information between one process with all other processes in its neighbour list is termed communication cycle. As such, information only travels to direct neighbours per communication cycle, which in turn requires multiple communication cycles to distribute information throughout the whole domain. However, information rarely travels from one end of the domain to another, while the multilevel neighbourhood structure allows us to limit the amount of cycles necessary in this case. It takes at most the diameter of the tree structure to reach all processes, in other words $2H - 1$, where H is the height of the tree. Additionally, non-continuous partitions, i.e. non-neighbouring nodes on a single process, spawn multiple information sources in a spatial sense.

The communication cycle may be designed in three ways. The simplest scheme is an ordered pattern. Each process traverses the list of neighbours in ascending order, first sending to and then receiving from all processes whose MPI-rank is smaller than their own. For all MPI-ranks greater than itself, a process first receives and then sends. Here, blocking `MPI_Send` and `MPI_Receive` calls are used. Figure 3.13 illustrates an example pattern with five participating processes, where every process communicates with every other process. Circles depict processes marked with their MPI-rank. Data exchanges are shown as arrows. Following this pattern, it takes seven pseudo steps to iterate through one complete communication cycle, with concurrent communication in steps three to five. Obviously, this pattern is not ideal because processes are occasionally idle, which is reinforced if some processors are slower or more burdened. In practice, it is sufficient and has shown good results [95], partly because full all-to-all communication never occurs.

A second version replaces the blocking send and receive routines with their non-blocking counterparts. This allows all communication to be processed in order of appearance, without having to wait for slower processes. The trade-off is the memory that has to be available for receive buffers, where for every remote process a individual buffer is needed. In the blocking case, a single reusable buffer is sufficient. A variant of this approach would be to use remote memory access (RMA) routines, i.e. MPI one-sided. Every process opens up a dedicated memory window for each neighbour process. The remote process then has remote access to this memory window without explicitly exchanging messages. One issue to consider here are the dangers of concurrent memory access, introducing additional synchronisation overhead. Frequent opening and closing of memory windows when neighbourhood relations change, is another.

The third and last method finds pairs of neighbour processes to communicate without order and without non-blocking communication routines to transfer the simulation data.

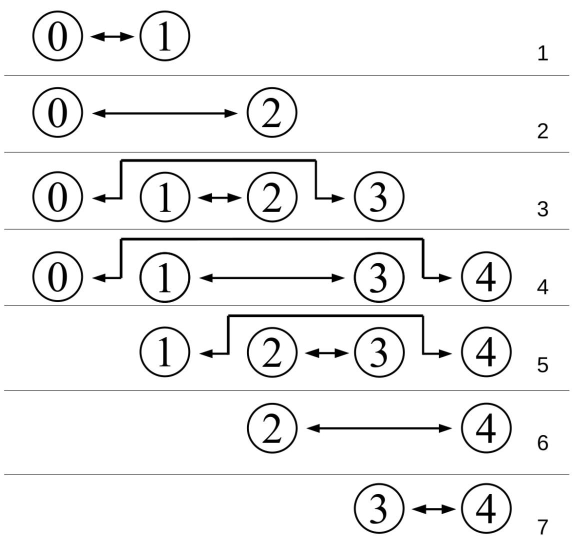
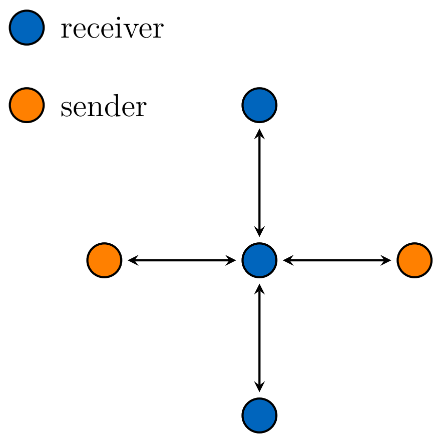


Figure 3.13: Ordered communication pattern with five ranks. Every rank needs to exchange data with every other rank.

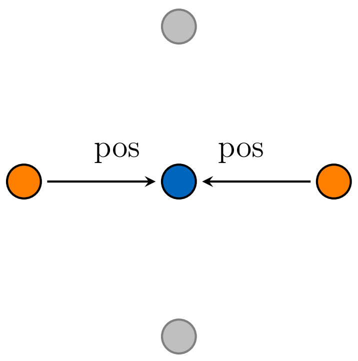
This saves memory since a single send and receive buffer is sufficient, as they can be reused immediately, similar to the ordered cycle. The advantage is that slower processes do not hold up communication as it can be completed in order of appearance. There is, however, the added cost of finding the pairs.

The outline of this algorithm is as follows. In a first step, shown in Figure 3.14a, each process randomly determines if it is a sender or a receiver and transfers this state using a non-blocking `MPI_Isend` to all its neighbours. A non-blocking `MPI_Irecv` is posted to acquire the state of all neighbours. In the second step, illustrated in Figure 3.14b, each sender checks if at least one receive is complete, and sends a positive response message to one of the receivers and negative responses to all other receivers. In the third step, shown in Figure 3.14c, a receiver sends back a positive response to one of the positive senders and a negative one to all other positive senders. The negative senders can be ignored. As soon as one sender is positive a pair has been found and actual communication can be established (Figure 3.14d). All other positive senders still need to receive a negative response, though this can be postponed. If a pair is found, or all remote processes have the same state, or all responses were negative, processes are assigned a new random state and start with the next cycle, trying to find a new communication partner (Figure 3.14e).

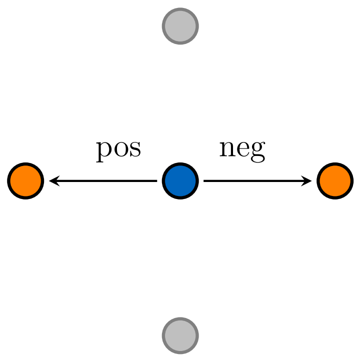
The last method should perform well when the load is badly balanced among the processes. Faster processes are able to complete their communication first, slower ones are granted additional time to finish their previous workload. Depending on the randomised sender and receiver configurations however, the amount of time it takes to find a pair varies highly and the added communication from finding pairs can be detrimental to the overall performance. Furthermore, every process needs to send and receive their state multiple times, even for pair searches that are already completed. Although MPI provides a facility to cancel outgoing messages with `MPI_Cancel`, its use is highly discouraged, especially for sends, as messages could already be partially send, making their cancellation more



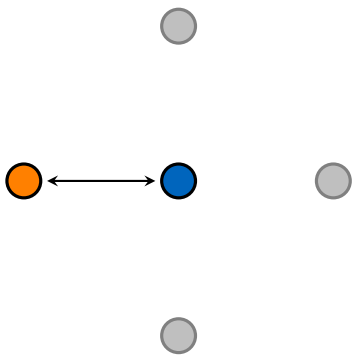
(a) Each process is randomly assigned to be a sender or a receiver. In the first stage, neighbouring processes exchange their state.



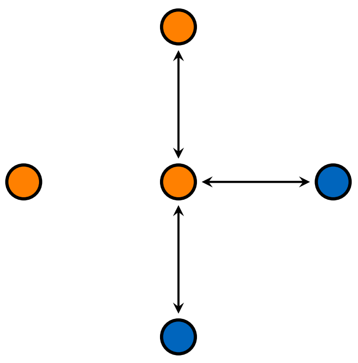
(b) Next, senders that have identified at least one neighbour to be a receiver, send a positive response to it. Other possible receivers would be send a negative response.



(c) The receiver picks a sender that has send a positive response and sends a positive response to it. All other senders are getting negative responses.



(d) A pair is found. Actual data exchange happens.



(e) Starting over with stage one. Processes are assigned new states and processes with already completed exchanges no longer communicate in this cycle.

Figure 3.14: Method to determine neighbourhood pairs in arbitrary order.

expensive as sending and receiving the actual message. Given a suitable partitioning and dynamic repartitioning, with mostly balanced loads among processes, the outlined method is not used in practice. If memory is not an issue, the second method, using non-blocking communication routines could be used. However, it again only performs better than the more memory efficient blocking version if the process load is highly imbalanced. Therefore, for all production runs, the ordered communication cycle that uses blocking communication routines is used in this research.

3.4 Domain Generation

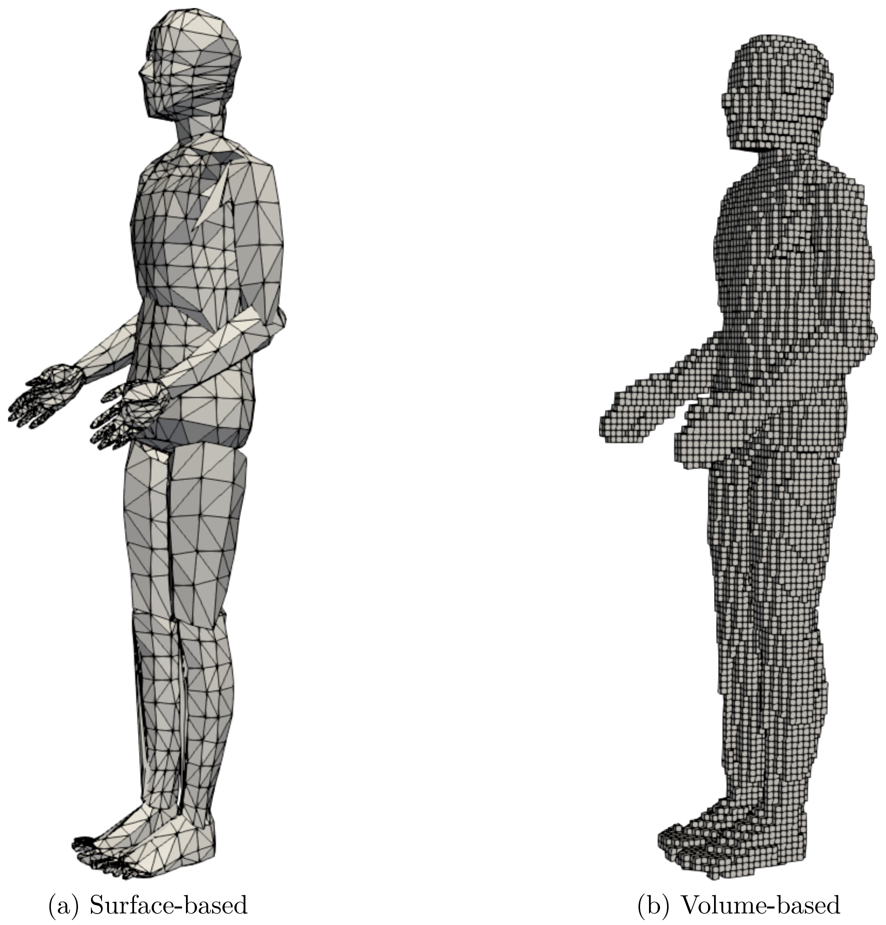
To set up simulation scenarios involving complex geometries, a fast and reliable way is needed to generate a volume-based model that fits the described data structure. Even more so, when scenarios should be evaluated in which objects are moving or the mesh is dynamically refined during runtime, which requires a re-evaluation of the decomposition on the fly. Consequently, the domain generation must be feasible on the target machine. Although a different machine could set up the computational domain with for example more lenient memory limits, during runtime it makes little sense to stop the application, port the current state to a different machine, where a new configuration is evaluated, before finally moving the new state back to the target machine. Therefore, the domain generation is not only subject to the limitations of the target machine, but also to the decentral data structure when it is being adjusted during runtime.

As mentioned before, the domain is generated following an adaptive space-tree decomposition. The user is able to define a minimum refinement depth to which the space-tree should be refined. Furthermore, regions of special interests can be defined where the domain tree should be refined further. Finally, the vicinities of geometric features are often the most interesting regions. To adaptively refine towards these features and to set boundary conditions accordingly, a space-tree-based voxel generator is used. This technique was first proposed by Samet [267]. Mundani further developed a fast method for various application scenarios [227] and Frisch adapted the method for his data structure [114].

3.4.1 Voxel Generator for Geometry-Based Decomposition

Primitives for this decomposition are simple voxels in three dimensions (pixels in two dimensions), with only spatial information about the physical size they occupy. The root voxel spans the whole computational domain. For every refinement, a set of child voxels is generated, their union occupies the same domain as their parent. The number of child voxels generated for each refinement again is dependent on the variable subdivision spacing. The algorithm uses a queue-based treatment of candidate voxels with the root grid as the initial element in the input queue. The decision whether a voxel should be refined is based on a range of geometrical intersection tests with a surface-based description that represents the domain geometry or features within. Commonly, a surface description using a stereolithographic format is used. The most well known data format of that class being the STL format. Here, the surface is represented through triangles.

Each voxel in the input queue is successively tested against more and more expensive intersections tests. If a triangle of the surface model is inside, touches or intersects the voxel, it is added to an output queue. The use of multiple intersection tests increases the efficiency of the algorithm, as the quick and inexpensive tests serve to discard many voxels early on, before evaluating against more expensive tests. After the input queue is empty, all voxels in the output queue are refined and their newly generated children are put in the input queue for the next pass until a prescribed depth has been reached, or no voxels are put into the output queue as refinement candidates during the intersection tests anymore. Figure 3.15a illustrates a surface-based model of a human used as the input for the voxel generator. In Figure 3.15b the volume-based results from the voxelisation are shown. The subdivision is equivalent to the classical octree scheme with eight children per refined voxel and a tree depth of $d = 6$. Shown are only the leaf voxels, i.e. the most refined ones that also contain geometric features.



(a) Surface-based model

(b) Volume-based model, depth $d = 6$

Figure 3.15: Surface-based input model and volume-based output model of a human for the voxel generator. The model is refined up to depth $d = 6$ using an octree scheme for subdivision.

To generate the computational domain using the voxel generator, the space-tree, which consists of all generated voxels is simply discretised. In other words each voxel becomes a grid of the data structure. To set the geometry boundary conditions, the leaf grids constructed from the voxels on the finest refinement level, i.e. the ones intersecting the geometry, are again subjected to the intersection tests employed by the algorithm. This time however, each individual cell of the respective grid is checked. If it contains, touches or is intersected by any surfaces of the geometric objects, geometry boundary conditions are set in the cell accordingly.

3.4.2 Decentralised Generation

In Frisch' implementation, a single central management instance is responsible for decomposing the computational domain and to assign each process an appropriate share of the data structure. This approach has the advantage that all topology information is stored centrally and allows for efficient global partitioning methods. However, it is limited by memory and does not scale. For problems with a static decomposition, which is only generated once, this method works quite well and if needed, can be computed on a different machine with more memory. In a dynamic setting however, both scalability and memory limits are crucial. Therefore, the present framework uses also a decentralised domain generation method to alleviate these bottlenecks.

In the present approach, all processes are responsible to generate their own share of the decomposition. Using one of the aforementioned space-tree decomposition methods, each process needs at least one unique root node to start the decomposition from. To achieve this, the domain generation is a multistage process. In the first stage each process independently generates an initial decomposition from a common root node, which represents the complete domain. Here, either a uniform space-tree decomposition or the adaptive voxel generator and an input geometry can be used. This initial decomposition must create a tree that has at least as many leaf nodes as there are processes participating in running the application. The generated tree is comparably shallow with respect to the final domain tree and has a low memory footprint, because only the structure is generated without initialising the corresponding discretised grids. Nevertheless, the depth and consequently the size of this initial tree is directly dependent on the amount of processes used and is therefore limited. For process counts on current supercomputers this is not an issue. This topic will be further discussed in section 3.4.5.

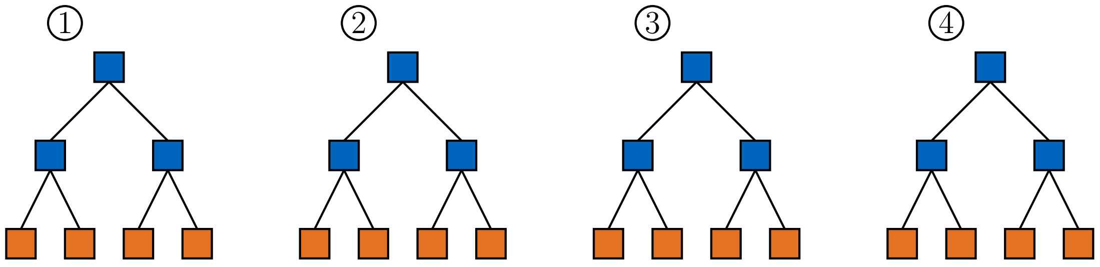


Figure 3.16: Stage one: Initial tree generation on all processes.

In Figure 3.16 the first stage is illustrated using a simplified example. Four process have generated an initial tree refined up to a depth of two, with four leaf nodes on the deepest refinement level in orange. The macrostructure connecting the leafs is shown in blue. So far, this approach has no drawbacks compared to a centralised generation. Instead of waiting idle, all processes simply generate the same structure.

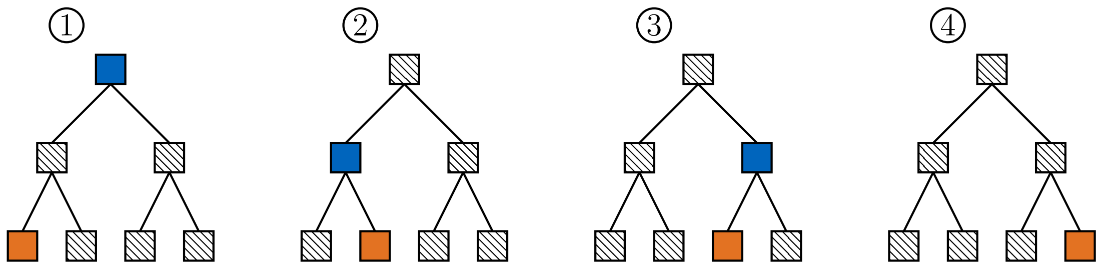


Figure 3.17: Stage two: Partitioning of the initial tree.

In the next stage, each process individually computes an initial partitioning of the current tree structure. As the complete structure is available, using an SFC linearisation on the hierarchical tree structure is convenient. But also all other global and local partitioning methods can be applied here. The only constraint that has to be adhered to, is that each process is assigned at least one of the leaf nodes of the tree. After computing the partitioning, it is virtually applied. That means, the UID of the nodes are changed to reflect their new rank and GID (see section 3.1.4. Afterwards, the neighbourhood metadata is updated with the new UIDs. This process can conveniently make use of the complete tree being present on all processes at this point. To conclude stage two of the domain generation, each process deletes all non local nodes determined by the partitioning. Figure 3.17 illustrates the configuration after stage two. Each process holds at least one leaf node (orange) and possibly a part of the overarching macrostructure (blue). Hatched nodes have been deleted and remain on other processes. Up to this point, every operation has been completely local and no communication took place. Nevertheless, the initial domain is partitioned and all neighbourhood metadata is up to date.

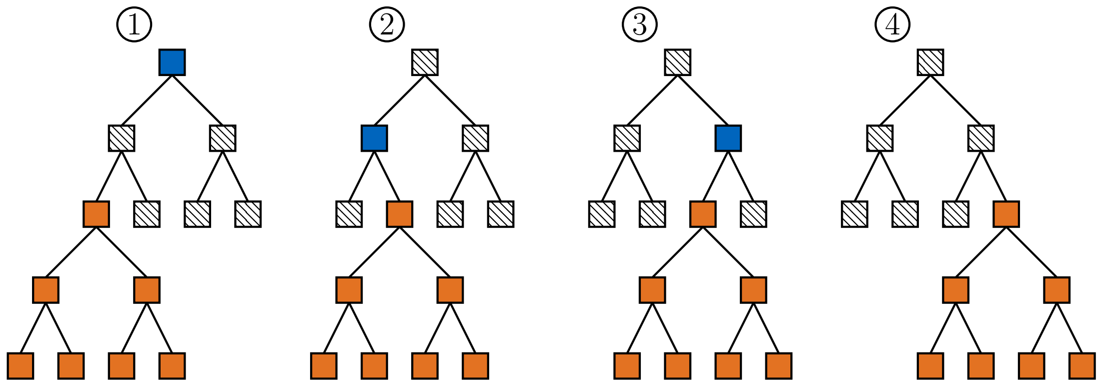


Figure 3.18: Stage three: Local subtree generation.

In the third stage, each process is now able to use one form of the space-tree refinement with one of its assigned leaf nodes as starting point. The generation of the individual subtrees has no dependencies at all and is therefore embarrassingly parallel. Figure 3.18 illustrates the current configuration in a simplified fashion. Each process has generated a local subtree from its assigned leaf node (orange).

3.4.3 Metadata Updates

The final step of the initial domain generation is to update the neighbourhood metadata of the nodes of the newly generated subtrees. In practice there are two possibilities. The first method is again completely local. Each process keeps neighbouring leaf nodes after the initial refinement and refines them as well. Now in addition to their own subtrees, a process also has the neighbouring subtrees available and can locally update the neighbourhood metadata. After updating, the neighbour subtrees are deleted. Drawbacks of this method are the higher memory footprint and a higher effort of having to construct also neighbouring subtrees.

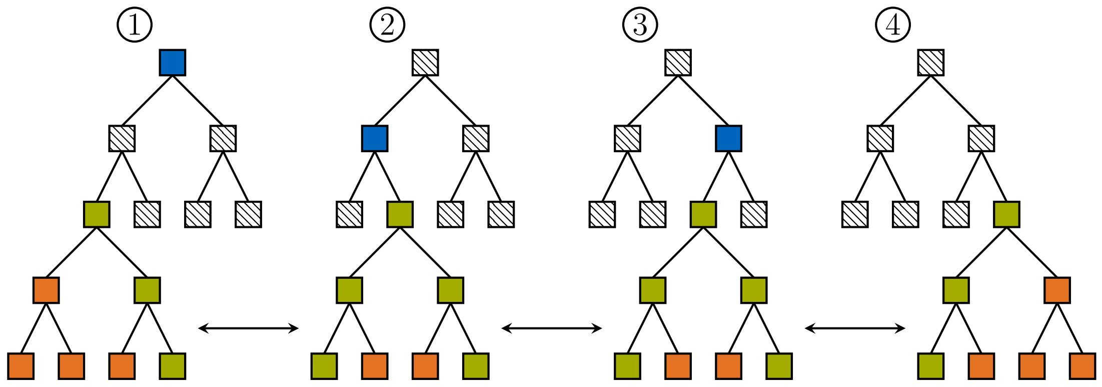


Figure 3.19: Stage four: exchange of surface trees.

The second is through information exchange with neighbouring processes. More specifically, communication has to be established with processes that hold spatial neighbours of the intermediate leaf nodes, from which the subtrees were generated. New neighbours can only lie on the surface of the subtrees opposing a processes' own subtrees. Therefore, the method only transfers this surface information of subtrees to neighbouring processes. The surface nodes of a space-tree still adhere to the tree structure. As such, a mock surface tree is generated on the receiving process that can be traversed exactly like the regular local tree structures. In Figure 3.19 the traversal of these surface information is illustrated. The nodes of the respective surfaces are depicted green and information to build up the tree structure is transferred to the appropriate neighbour process.

Transferring the surface trees and also space-trees in general can make use of very efficient memory storage schemes also known as succinct data structures [166]. In turn, the messages length to send these trees is minimal. For binary trees, one possible succinct structure is generated through converting the tree structure into a list of bits who encode the tree traversal. The framework uses a depth-first traversal, whereas every bit encodes whether a node is refined or not. With the knowledge of the subdivision spacing, i.e. the amount of new nodes generated when a refinement takes place, the tree can be rebuilt from this list of bits. In Figure 3.20 an example tree and its bit representation is illustrated. The subdivision per refinement is two. The list is generated going through the tree from top to bottom first and from left to right second. The *id* is an increasing number, signifying the order of traversal of the tree. *ref* takes the value 1 if the node is refined and 0 otherwise. To transfer the surface tree from one process to another, one

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

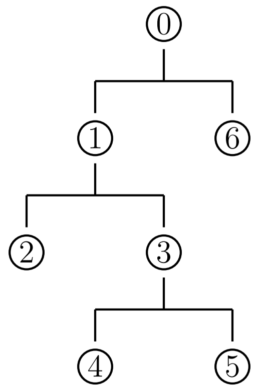


Figure 3.20 & Table 3.1: Depth-first encoding of an example space-tree (binary-tree) with seven nodes in total. *id* refers to a consecutive numbering of nodes following their traversal, *ref* is encoding the tree structure and encodes whether a node identified by *id* is refined (1) or not (0).

sends the bit list followed by a list of unique identifiers (the UIDs) in the same order as the bits reference the corresponding nodes. This information is sufficient to build a mock tree representing the neighbouring subtree surface on the target process.

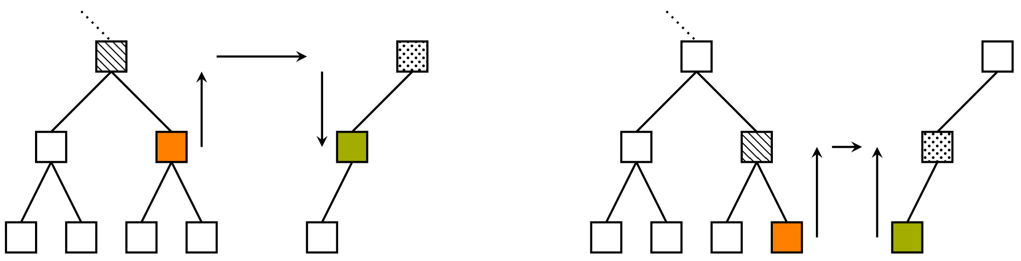

(a) First refinement level.
(b) Second refinement level.

Figure 3.21: Multistage neighbourhood reference update using a bottom-up, horizontal and top-down local tree traversal.

The method to update neighbourhood metadata is illustrated in Figure 3.21 and is explained as follows. Neighbourhood references are updated top-down starting with nodes on the first refinement level of a subtree. In Figure 3.21a the subtree on the left and a mock surface tree on the right are present. The neighbourhood reference of the orange node is updated by traversing the tree upwards to the root node of the subtree (hatched). The metadata references of this node are up to date from a previous metadata update of the initial tree, therefore its neighbour on the mock surface tree can be queried (dotted). From here, using the location information in the UID, the respective child node is found (green). After all neighbourhood references on the first refinement level have been updated, the procedure is repeated on the next refinement level. With the now updated references on the coarser level, the traversal of the two trees again only includes two levels. This is illustrated in Figure 3.21b, were the reference update queries the neighbour of the now updated node from the previous illustration. The same method to update neighbours is used in the local update method, with the surface information being available in the complete neighbour subtree.

The framework uses the later method of exchanging the surface trees and updating neighbourhood references from them. The surface trees are likewise needed to resolve the 2:1 tree balance constraint, which is addressed in the following section. Since they have to be transferred in any case, their use for the neighbourhood updates is the sensible choice.

3.4.4 2:1 Tree Balancing

As explained in section 3.1.2, the framework supports ghost cell updates across different refinement depths. Nevertheless, these exchanges always interpolate values to match the discretisation between grids and therefore introduce an error. To ensure the convergence of the numerical solvers, the amount of refinement level discrepancy between spatial neighbouring leaf grids is limited to at most one level of difference. For the initial domain generation, this is achieved through balancing the domain tree before discretising the nodes. The balance is established solely trough refinement, coarsening of previously refined nodes is not used to balance the tree.

Balancing space-trees is a well known problem in literature. The interested reader is referred to [164], [203] and [296] for publications detailing the problem. The decentralisation of the data structure makes this problem more involved however. First, the method to balance a single space-tree is illustrated.

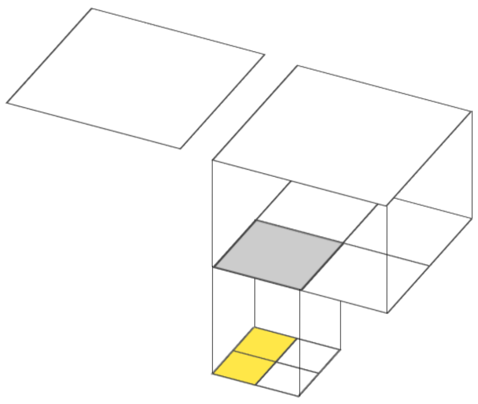


Figure 3.22: Domain configuration with a violation of the 2:1 balance constraint. A grid has spatial neighbours on the same level, one refinement level down, marked in gray and two refinement levels down, marked in yellow.

Figure 3.22 illustrates a configuration where a grid has two spatial neighbours two refinement levels further down, marked in yellow. This configuration therefore violates the 2:1 balance constraint and the tree node representing the original grid has to be refined. The method is constructed from the point of view of the nodes subject to be possibly refined. For every leaf node, their geometrical neighbours are referenced. From them, their children on the opposing side of the original grid are visited. If any of these children are refined themselves, the method found a violation of the constraint and marks the original node for refinement. After the node has been marked for refinement, all further checks can be relinquished and the next leaf node is addressed. The pseudo code for the method is also given in Algorithm 1. This balancing method is repeated either $H-1$ times, with H being the height of the subtree or until a complete iteration through all current leaf nodes does not detect a single violation of the constraint. A simple flag, set when a node is refined and unset between iterations of the method, is sufficient for this task.

Algorithm 1: Local tree balancing method.

```
foreach leafNode do
    foreach leafNode.spatialNeighbour do
        if leafNode.spatialNeighbour is refined then
            foreach leafNode.spatialNeighbour.childNode do
                if leafNode.spatialNeighbour.childNode is bordering leafNode and is
                  refined then
                    refineNode ( leafNode );
                    goto nextLeaf
                end
            end
        end
    end
    nextLeaf;
end
```

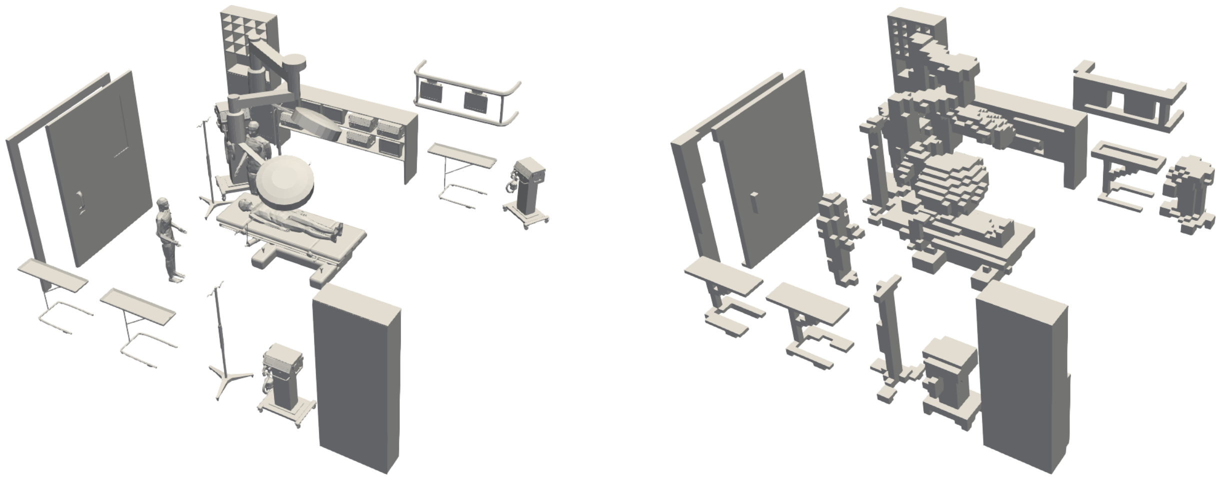
This method is used to balance the initial tree on all processes and to balance the local subtrees. However, balancing subtrees across processes requires information about the neighbour subtrees. Again, the surface of the neighbouring subtrees is sufficient to balance the local subtrees as part of the global domain. Therefore, the same mock surface trees are used in balancing the domain and updating the local subtrees. The cycle of sending surface information, building mock trees and using them has to be repeated again $H-1$ times as in the pure local method as newly generated nodes warrant subsequent updates and refinements on remote processes and vice versa. After the first cycle though, only the newly generated nodes have to be exchanged, not the complete surface anymore. Breaking early is however not possible this time. Refinements may cascade through a remote subtree caused by updated tree surfaces on the opposite side of the neighbour subtree (see also section 3.5.1). Refinements caused by nodes on the opposite side of a remote subtree may only be visible after refinements have cascaded through the remote subtree in a later cycle, with no violations of the balance constraints in between.

To conclude the method one further remark has to be made. Nodes generated to satisfy the balance constraint cannot have geometry boundary conditions. If a node had parts of the geometry in the first place, it would have been refined previously, when the subtree was generated. Therefore, intersection tests to evaluate geometry boundary conditions in the cells of the discretised grids are not necessary.

The idea to also generate remote subtrees from neighbour leaf nodes of the initial tree as a means to balance in a pure local fashion, similar to the method proposed when updating neighbourhood references, can not be applied here. To evaluate cascading refinements caused by a second level neighbour subtree (the neighbour of a neighbour), would require knowledge of the subtrees of the second level neighbour as well. To evaluate all cascading refinements, all subtrees of processes with neighbour connections over $H-2$ intermediate processes would have to be considered. This strategy is directly opposing the decentral design and was therefore not considered. Furthermore, the advantages over a complete centralised domain generation are vanishing, even more so for large domains.

3.4.5 Performance Evaluation of the Domain Generation

To show the viability of the decentral generation approach, a strong scaling study on a representative example was performed. All tests were run on Leibniz Supercomputing Centre's (LRZ) Linux Cluster System [201] on the CoolMUC-2 cluster segment [200]. This segment consists of 812 28-way Intel Xeon E5-2690 v3 Haswell-EP nodes with Infiniband FDR14 interconnect and two hardware threads per physical core. The theoretical peak performance of the segment is 1,400 TFlop/s. Furthermore, the compiler used was the Intel Compiler in version 19.0.



(a) Surface-based model (b) Volume-based model, depth $d = 7$

Figure 3.23: Surface-based input geometry files and voxelised domain. Visualised are the most refined grids on depth $d = 7$.

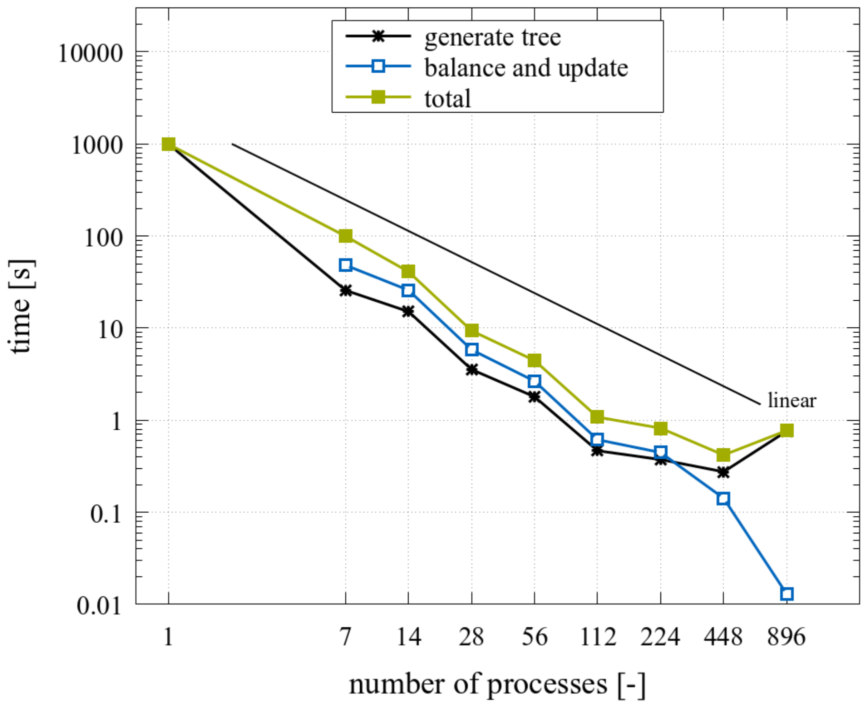
The input file for the voxel generator contains the geometry for an operating theatre located at the university hospital “Klinikum rechts der Isar” in Munich. It has been used as geometry for fluid simulations in [326, 248, 113]. In addition to the operating theatre's furnishing, two models of doctors as well as a patient have been added. The model has dimensions of 6.3 m × 6.25 m × 3.5 m, and in total all input files contain 84,072 triangles. Figure 3.15 shows both the combined input surface files on the left and the voxelised volume-based model on the left. The later depicts only the most refined grids on depth $d = 7$. The complete computational domain consists of a little more than 121,213 nodes, whereas each node is discretised using a grid with 8 × 8 × 8 × cells.

The domain generation was conducted to depth $d = 6$ and depth $d = 7$ with increasing process counts, starting from a single process on one cluster node up to 896 processes, using 32 cluster nodes with 28 processes each. Each refinement of a tree node generates eight child nodes, using a bisection of the domain in every dimension, representing an octree structure. Each individual test combination was run multiple times to calculate the truncated mean, where extreme outliers were discarded. The Linux Cluster is a shared resource that is used by many researchers simultaneously. Therefore, outliers are caused by other running applications that also use the interconnect. Consequently, without the outliers, the given truncated mean agrees very closely with the minimum observed times.

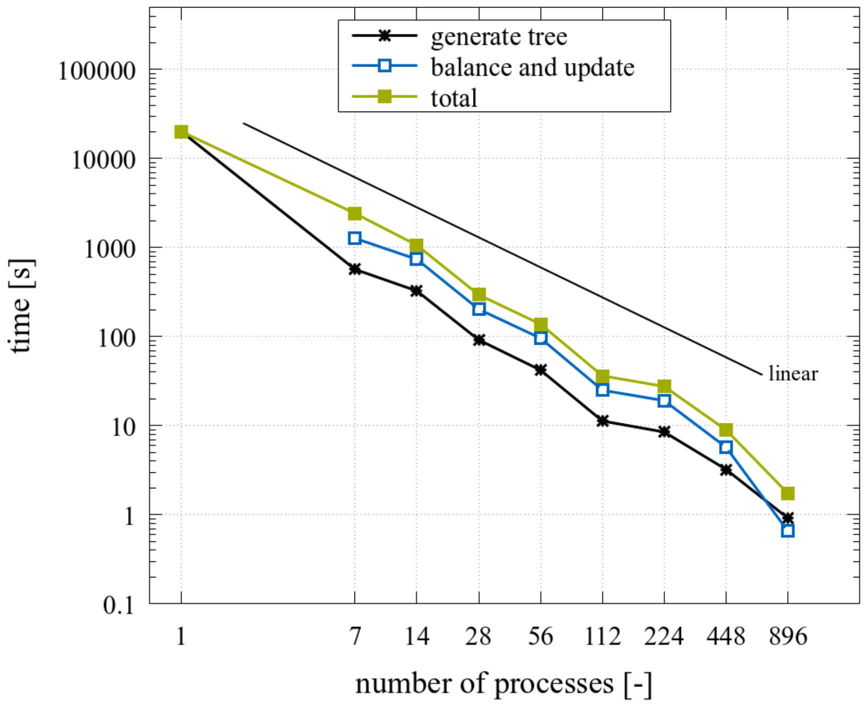
The number of used processes determines the height of the initial tree generated in stage one of the domain generation procedure. In this test case, a simple uniform refinement for the initial tree was used. A single process does not need an initial tree, for seven processes an initial tree refined to depth $d = 1$ with eight leaf nodes is needed. For 14, 28 and 56 processes, the intermediate tree needs to be refined to depth $d = 2$ with 64 leaf nodes. For 112, 224 and 448 processes the tree is refined to depth $d = 3$ and 512 leaf nodes and finally, using 896 processes requires an intermediate tree refined to depth $d = 4$ with 4,096 leaf nodes. The measured times are split into two categories. The first combines stages one to three of the domain generation. That means it contains the generation of the initial tree, virtual partitioning using an SFC linearisation, update of local neighbourhood metadata and the generation of the local subtrees. Also included are the intersection tests to set the boundary conditions on the cells of the leaf grids on the finest refinement level. The initial tree does not have to be balanced as it is uniformly refined. The second measurement includes the local subtree balancing and the communication and building of the mock surface trees.

In Figure 3.24 the measurements are shown. For both refinement depths, a steady decline is observed in both generation times and balancing effort. For refinement depth $d = 6$ (Figure 3.24a), the complete domain setup takes roughly 1.000 seconds on a single process and 0.9 seconds on 896 processes, whereby the lowest total time is measured at 448 processes used. From 448 processes to 896 processes used, the time for the refinement stage increases again. At 448 the initial tree is refined on every process up to depth $d = 3$ resulting in an average number of leaf nodes as starting points for local trees per process of 1.1. For 896 processes the initial tree must be refined one level further, which results in an average number of leafs per process of 4.6. The added time can therefore be explained by the added effort of uniformly refining the intermediate tree one level further, generating more remote nodes that are deleted anyway. Moreover, the amount of subtrees to generate is higher as well. Balancing the subtrees becomes cheaper though, as the subtree height is lower. This also explains the increased speedup for the balancing stage between.

For refinement depth $d = 7$ (Figure 3.24b), the domain generation takes roughly 20,000 seconds on a single process and two seconds using all 896 processes. On a single process, the initial tree is equivalent to the final tree. Therefore, balancing and updating the subtrees is not necessary. The tree generation clearly benefits from the parallelisation. The balancing and update routines, even though they are only necessary because of it, also benefit from a higher degree of parallelisation.



(a) Refinement depth $d = 6$



(b) Refinement depth $d = 7$

Figure 3.24: Times for domain generation using the decentral domain generation procedure. The measurements are broken down into generating the initial and the local subtrees, and secondly into balancing and updating the subtrees. The times are measured against the amount of processes used from 1 to 896.

The measured speedups for depths $d = 6$ and $d = 7$ are on average four times the number of processes used. This superlinear speedup can be mainly attributed to cache effects. Storing the complete domain tree on a single node far exceeds the size of faster caches. Increasing parallelisation not only shares workload, but also splitting up the domain tree allows for subtrees to fit into cache memory and allows faster access. Furthermore, increasing the process count leads to less local leaf nodes and therefore less subtrees to balance, if the depth of the initial tree is not increased simultaneously. In the best case, one less subtree saves communication with six neighbouring processes in three dimensions. If, on the other hand, the initial tree has to be refined one level further, the height of the subtrees is one less. This saves a complete balancing cycle.

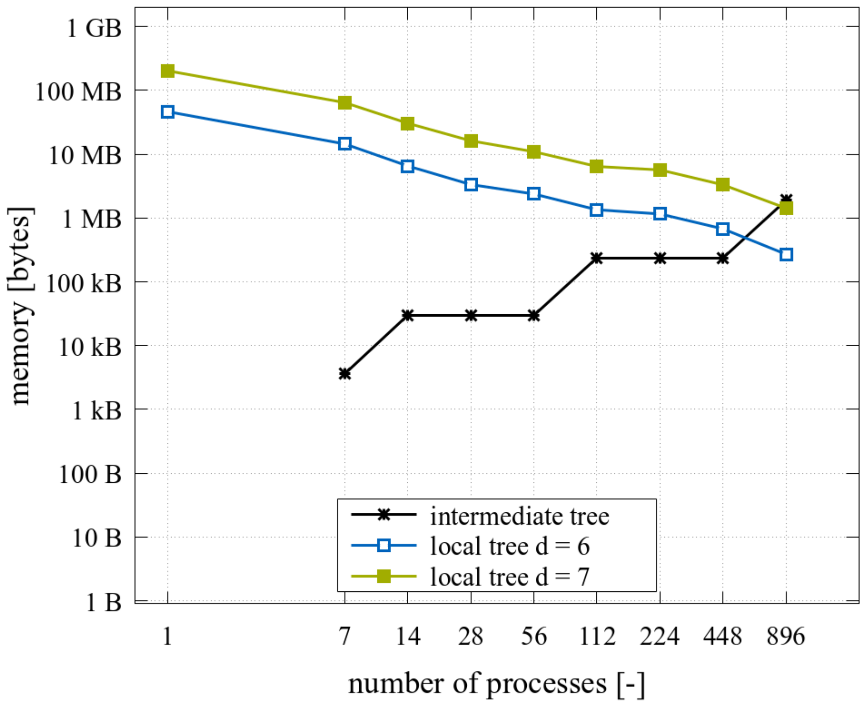


Figure 3.25: Memory requirement of the intermediate tree and the complete node share of the process with the most nodes for depths $d = 6$ and $d = 7$ measured against process counts from 1 to 896.

To conclude the measurements, Figure 3.25 illustrates the memory consumption of the initial tree and the local subtrees. As mentioned before, on a single process, the initial tree is not needed and the local subtree is the complete domain tree. For refinement depth $d = 6$ the tree contains 27,770 nodes which consume 45 MB. For refinement depth $d = 7$ the tree consists of 121,213 nodes. The memory consumption is 200 MB respectively. With more processes holding exclusive parts of the domain, the memory requirement decreases. With 896 processes participating, the subtree memory requirement on a single process is 270 kB for depth $d = 6$ and 1.4 MB for depth $d = 7$. Naturally, the memory consumption of the initial tree is increasing with its height. The intermediate tree for 896 processes refined to depth $d = 4$ with 4,681 nodes in total requires roughly 1.9 MB. However, after the initial tree has been partitioned, foreign nodes are deleted, which brings the memory requirement down to roughly 2 kB.

As the size of the initial tree is directly dependent on the amount of processes used, it is the limiting factor of this approach. When using 896 processes the memory requirement of the initial tree outgrows the decreasing requirement for the local subtrees. At approximately half a million processes, the initial tree would have to be refined to depth $d = 7$, matching the single process requirement. Even if this example is unrealistic, it shows the limitations of this approach. The supercomputer Fugaku incorporates 7.6 million cores with 660 MB memory per core. An initial tree refined uniformly to depth $d = 7$ uses 960 MB of memory. The approach therefore allows only the usage of less than half a million cores or the initial tree has to be adaptively refined, whereby the geometry determines the number of nodes and therefore the memory consumption. Summit, the current number two supercomputer with 2.4 million cores and 1.2 GB memory per core on the other hand is able to use the approach and utilise all available cores.

3.4.6 Concluding Remarks for the Decentral Domain Generation

This concludes the decentral domain generation procedure. Within the constraints of the decentral framework, the procedure succeeds in lowering the time it takes to generate the domain compared to a centralised approach. Furthermore, the memory requirement on a single process has been lowered in the observed range. Nevertheless, the initial tree acting as a starting point of an individual subtree refinement poses limitations as its height and consequently its memory requirement are directly connected with the number of processes used. This limitation comes close to the capabilities of current supercomputers and occasionally prohibits the use of all their resources. A conclusive solution to the increasing sequential part has yet to be found.

Another important remark is that the ensuing domain configuration is not yet load balanced. Only the initial tree is partitioned. Depending on the configuration of the geometry in the input file or the user supplied domain configuration, the complete partitioning after the subtrees have been generated might be highly imbalanced. This results in a high variability of workload per process. At this point, global partitioning methods are no longer feasible without gathering the complete tree structure on a single repository. As such, the framework employs a local diffusion approach supported by the decentral data structure to ensure a proper load distribution, while also minimising inter-process communication and minimal redistribution costs. The decentral partitioning is detailed in chapter 4.

A similar strategy, where each process independently generates the domain, has been utilised recently by Jomo [171, 170], within his work, where he addresses various concepts to parallelise the finite cell method. In his domain generation method, all processes generate the complete domain independently and retain the ensuing structure. A partitioning is realised by simply marking active and inactive parts of the local domain, where the decision which parts remain active is forwarded to the Zoltan library [29]. As mentioned before, Zoltan provides different geometric and graph-based partitioners.

3.5 Adaptive Mesh Refinement and Coarsening

Adaptive mesh refinement (and coarsening), abbreviated AMR, is a method to adaptively change the resolution and therefore the solution accuracy of specific regions within the computational domain during runtime of the application. This is especially useful for highly dynamic simulations with fast varying localised effects. The concept has become widespread in modern numerical simulation and has seen use in many applications areas. Examples are astrophysics simulations like the simulation of the formation of stars and many kinds of fluid flow phenomena which can be described by Shallow Water Equations and Navier Stokes Equations [25, 24].

The Chair of Computational Modeling and Simulation [63] (formerly Chair of Computation in Engineering) of the Technical University of Munich has a long history of research into adaptive and runtime adaptive methods, especially with respect to FEM and FCM. In the 1980s, Rank worked on error estimators as an indicator for refinements in the *hp* adaptive version of the FEM [258, 260]. More recently, work has been done to extend the FCM with hierarchical *hp-d* adaptivity for local mesh refinement by Schillinger [275]. Moreover, Zander et. al. proposed a new refinement strategy termed multi-level *hp*-method to address the problem of hanging nodes in dynamically changing domain discretisations [341, 340]. Lastly, Kopp et. al. have again extended the multi-level *hp* framework to include arbitrary dimensions. This allows them to not only address spatial adaptivity, but also include adaptivity in temporal direction within their Galerkin framework [188]. Their framework has been successfully applied to transient problems with dynamic AMR and for additive manufacturing applications [187, 186].

The runtime adaptive mesh approach has a number of advantages over a fixed mesh. In general, the computational costs and the storage requirements are less than with a static mesh. The runtime adaptive mesh evolves and increases the resolution in regions with a higher demand on accuracy, while it likewise decreases the resolution in regions with a lower demand. A static approach has to discretise every region of the computational domain with the highest resolution required during the runtime. As regions do not always require this level of effort, resources are wasted when the accuracy demand is low. Furthermore, to estimate which regions need which resolution during the runtime requires a large amount of a priori knowledge of the evolution of the solution. Bad assumptions can lead to bad results due to low accuracy or high costs due to high resolution in regions where the accuracy is not needed. An adaptive approach reacts to the evolution of the solution during the runtime and requires less knowledge beforehand.

The frameworks' data structure is designed to easily incorporate AMR capabilities. A local refinement is simply achieved by refining the node of the domain tree representing the domain where the additional accuracy is desired. In other words, a grid is refined through the generation of a set of children grids. As the amount of mesh points per grid is equal among all grids, regardless of the domain they discretise, more grids discretising the same domain result in a higher amount of mesh points and consequently a finer resolution, better accuracy and allows to better capture local phenomena. Conversely, coarsening is achieved by “unrefining” a node. Meaning the deletion of the child nodes, respectively the children grids. The solution procedure is entirely unaffected by a refinement or a coarsening. All stencil operation are confined to a single grid, and the communication procedures to communicate values between grids support data exchanges over various refinement depths.

Important for an effective runtime adaptive mesh is the criterion used to issue refinement and coarsening tasks. A viable choice is refinement towards geometry, especially if it is moving through the domain. In CFD simulations, like the solution of Shallow Water Equations, mesh refinement towards moving wave fronts, i.e. towards high water level gradients can be used. Depending on the problem, refinement towards regions of high pressure gradients, high energy dissipation in turbulent regimes and towards high vorticity to resolve vortex phenomena are also good choices for general Navier Stokes problems. Another possibility are problem dependent error estimators, which can hint at high error regions, where a higher accuracy is needed, or regions with lower errors, where the resolution could be decreased. Yet, good error estimators may be very costly and lead to diminishing returns of the advantages of the mesh adaptivity. Verfürth gives an overview over error estimators in the context of adaptive mesh refinement [313]. Other references with respect to error estimators can be found for example for finite difference methods in [182], for finite volume methods in [47, 6] and for finite element methods in [3, 128, 259].

Refinement, coarsening and deletion operations may cause violations of the 2:1 balance constraint (see section 3.4.4). Refinements are always possible, may cause subsequent refinements tough. Coarsening and deletions can be rejected when they result in a violation of the constraint. Furthermore, processes cannot end up with zero nodes, respectively discretised grids. As mentioned before, the inter-process communication cycle includes processes with neighbouring connections. Without any nodes, processes won't be able to communicate anymore.

All refinement, coarsening and deletion operations cause metadata updates. After refinements, the new local grids need to set their neighbour references. Remote grids that have gotten new neighbours must be informed to register the new neighbour references. Similarly, grids must be informed when neighbours have been deleted to invalidate the appropriate neighbour references.

To send different tasks in the AMR pipeline between processes so-called queries are used. These queries are structured similar to the unique grid identifier described in section 3.1.4. The MPI-rank of the origin grid – the grid which issues the update – and remote grids are implicitly known by the ranks of the sender and receiver processes. The GID refers to the affected or target grid. Furthermore, a query contains a `task` field, to encode the task in the current communication cycle, a `side` field, encoding a reference to the direction of the origin, and finally the hash if the task applies to one of the targeted grids' children. This allows the unique identification of the query task as well as both the origin and the remote grid using a single 64 bit integer.

The following sections cover the intricacies of the refinement, coarsening and deletion methods with emphasis on the compliance of the balance constraints and the subsequent metadata updates.

3.5.1 Refinement

A refinement increases the local resolution of the simulated domain. The same domain that was represented by a single node, discretised by a grid, will be represented by a set of nodes discretised with grids of equal size after a refinement. The amount of child nodes can be freely chosen. The implementation described here allows subdivisions for the discretised domain from 1 to 7 in each cardinal direction. A node may only be refined if it is not yet refined. Furthermore, it cannot be partially refined. Either it is refined and the full set of child nodes exist or none of them do. The decision as to whether a node should be refined is made by the process that holds it and depends on the requirements of the problem, as elaborated earlier.

Transferring values from coarser to finer grids is implemented in two ways. A simple switch allows to change between the two. As the domain occupied by the parent exactly matches the domain occupied by all its children, the first method simply replicates the same values from the parent into all child grids. However, this leads to unphysical oscillatory behaviour when calculating the second derivative, for example. A better method is to interpolate the values from the coarser to the finer grids, using bilinear in two and trilinear interpolation in three dimensions. At the boundaries of the fine grids, the values in the ghost cells of the parent grid are taken into account to interpolate the values.

Refining a grid can lead to a violation of the 2:1 balance constraint, which requires leaf grids discretising neighbouring domains to be on refinement depths with a difference of one level at most. In other words the interpolation between ghost layer exchanges during the solution process is limited to one refinement level. A refinement will never be rejected due to a violation of the balance constraint. In contrast to the tree balancing procedure, where the leaf nodes themselves evaluate whether they need to be refined, a consecutive refinement query is issued by the node about to be refined. As the tree structure is completely distributed, running several balancing cycles is more expensive in contrast to the balancing of local subtrees and the cheap transfer of surfaces of neighbouring trees.

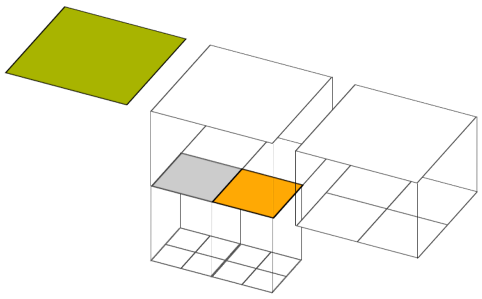


Figure 3.26: Domain configuration where one refinement (gray) leads to a violation of the 2:1 balance constraint and warrants the green grid to also refine. A refinement of the orange grid does not violate the balance constraint.

To answer the question whether a refinement breaks the one level difference constraint and warrants another refinement, a grid simply checks if it has spatial neighbours on all sides. If it is missing a neighbour on a side, a refinement will break the 2:1 balance. Figure 3.26 illustrates in which configurations a refinement breaks the balance constraint. The grids marked in gray and orange have been refined. The orange grid has valid neighbours in east, west and north direction (the south neighbour is assumed). The gray grid is missing a neighbour in west direction. The refinement therefore violates the balance constraint. To re-establish the balance, the green grid, i.e. the west neighbour of the parent of the original grid has to be refined as well.

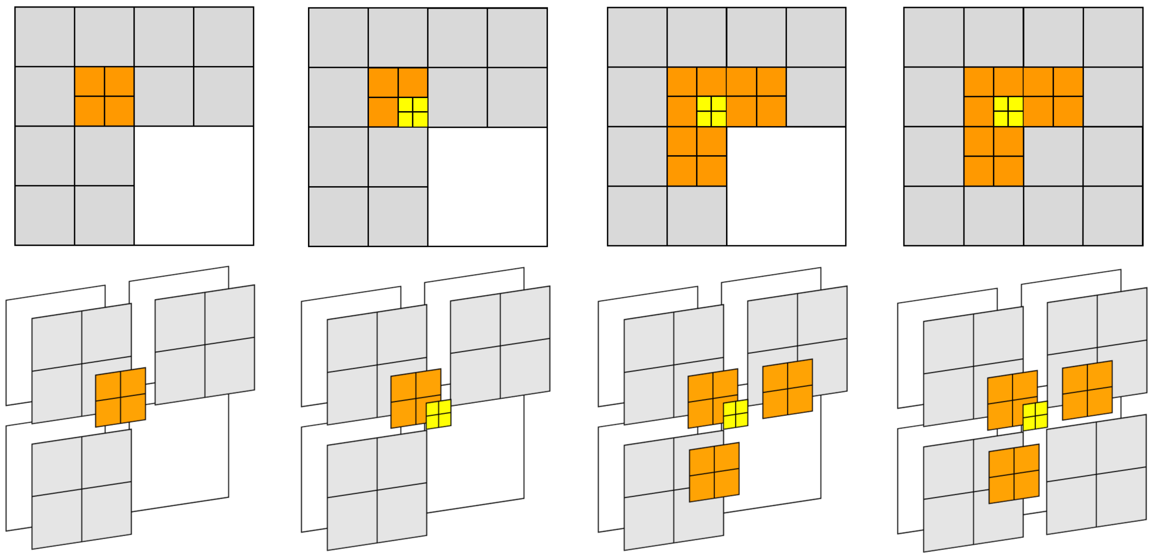
Algorithm 2: Pseudo code for the refinment info method, receiver side.

```
refinementInfoIncoming ( )

    receiveQueries ( );
    foreach query do
        if queryOrigin is a geometric neighbour and queryTarget is not refined then
            addTargetToRefinementCandidates( queryTarget );
        else if queryOrigin is child and neighbour is local and not refined then
            addTargetToRefinementCandidates( queryTarget );
        else if queryOrigin is child and neighbour is remote then
            addNewQuery ( queryTarget.neighbour );
        end
    end
```

From an implementation perspective, all processes evaluate violations of the balance constraint, which can be resolved locally first. This is the case when the parent of the refined grid as well as the neighbour of the parent are local and can be refined to re-establish the balance. If the parent or the neighbour of the parent are remote, queries are generated to be send to the processes holding the affected grids. The procedure on the sender side bundles these queries for their respective targets, so they can be sent in a single message. The method to inform remote processes about a refinement on the receiver side is illustrated in pseudo code in Algorithm 2. Using the GID information the receiver process determines the target grid of the query. The side information allows to determine the origin grid. If the origin is a spatial neighbour and the target grid is not refined it must be refined. If the target grid is the parent and its neighbour is local and not refined, the neighbour can be directly refined as well. If the target grid is the parent and its neighbour is remote, a new query is generated to be forwarded in the next communication cycle.

If the origin grid, its parent and the parent's neighbour grid are held by three different processes, the refinement query must be forwarded once, resulting in two total communication cycles. Furthermore, a refinement may cascade through the domain. Figure 3.27 illustrates a refinement cascading to the maximum amount of consecutive refinements possible in two dimensions. The initial structure is shown in Figure 3.27a. The domain is adaptively refined twice. Grids on depth $d = 0$ are shown in white, grids on depth $d = 1$ in grey and grids on depth $d = 2$ in orange. At this point, the domain balance is satisfied, with no depth difference greater than one between spatial neighbours. In Figure 3.27b, the right lower grid on depth $d = 2$ is refined one level further, which generates an imbalance. Grids on refinement depth $d = 1$ border the new ones on depth $d = 3$. Since the origin grid of the refinement does not have a spatial neighbour in the east and bottom direction, the parent, and subsequently its spatial neighbours in the respective directions, must be



(a) Initial structure (b) Refinement (c) First cascading refinement (d) Second cascading refinement

Figure 3.27: Cascading refinement in two dimensions. Colour coding signifies the refinement depth from lowest to highest, white, grey, orange and yellow. Top row depicts an overlay of all grids with the most refined representation of the domain visible. The bottom row shows all grids, including coarser representations of the domain.

informed. The first refinement cascade results in the state illustrated in Figure 3.27c. Again, an imbalance is observed. This time caused by the last refinement. Likewise, the parent of the origin grid and its neighbours in the respective directions must be informed and refined. The final balanced state is depicted in Figure 3.27d. This example shows that a refinement may cascade once in each cardinal direction. In the worst case, where all grids (origin, parents and neighbours) are on different processes, each forwarding requires two communication cycles for the information to be propagated, four communication cycles are needed in two dimensions and six in three dimensions to deal with all possible cascading refinements.

3.5.2 Coarsening and Deletion

A coarsening decreases the local resolution of the domain. A grid that is refined is said to be coarsened when its children grids are deleted, leaving the domain to be represented only by a single grid instead of multiple ones. The framework only allows one level of coarsening per iteration through the AMR pipeline. This means that a grid that has just been coarsened cannot be deleted in the same AMR cycle. As discussed for refinements, a grid cannot be half refined respectively half coarsened. For coarsening this means all children, without exceptions must be deleted when a grid is coarsened. If one or more of the children to be deleted are themselves refined, they cannot be deleted and the original coarsening is rejected. Again, just as with refinement, the decision as to whether a grid should be coarsened is made by the owning process. It aims to keep the accuracy of the solution within acceptable bounds while increasing the speed of the application.

There are three reasons which lead to a rejection of a coarsening. Grids cannot be coarsened if one or more children are refined themselves. Furthermore, empty domains cannot be allowed. In other words, processes cannot end up with zero assigned grids after deleting a set of child nodes. As discussed in section 3.2, the neighbourhood model considers processes holding grids with neighbourhood relations to be neighbouring processes. Processes without neighbourhood relations do not participate in the communication cycle anymore and therefore cannot be targets for grid migrations within the dynamic repartitioning. Consequently, these processes would be idle for the rest of the runtime, wasting valuable resources.

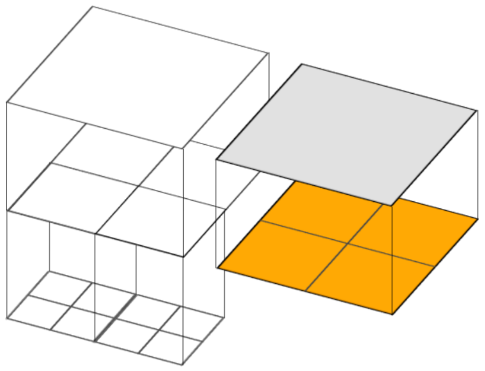


Figure 3.28: Domain configuration where the coarsening of a grid (grey) leads to a violation of the 2:1 balance constraint. To carry out the coarsening the children in orange will be deleted, leading to a discrepancy of two refinement levels between spatial neighbours.

The last reason for a rejection of a coarsening results from a violation of the 2:1 balance. This happens when the geometric neighbour of a grid to be deleted is refined. Figure 3.28 illustrates a configuration in which the coarsening of a grid (grey) must be rejected because it would lead to a violation of the constraint. Coarsening the grid entails deleting its children, marked in orange, which leads to a discrepancy of two refinement levels between spatial neighbours. To detect this violation two possibilities are conceivable.

The first comes from the perspective of a grid about to be deleted. One could send a query to all its spatial neighbours, asking if at least one of them is refined. The second possibility is from the perspective of the grid to be coarsened. This grid sends a query to its neighbours, who forward the query to their children, again asking whether they are refined. In the framework the first method is implemented. To determine whether the first reason for a rejection is true, that is whether one of the children to be deleted is refined, communication between the grid to coarsen and its children to be deleted has to be carried out regardless. As such, the implemented method communicates less and is therefore preferred. To evaluate all coarsenings five communication cycles are needed in the worst case. This worst case occurs when the grid to be coarsened is held by one process, any of its children are held by a second and any of the neighbours of these children are held by a third processes. A communication cycle is needed to send the initial coarsening query from parent to children, a second cycle forwards the coarsening query, a third cycle receives the response about the possible violation of the domain balance. This response must be sent back to the original parent grid in a fourth cycle. After all responses have been evaluated, the coarsening is rejected or confirmed. If the coarsening has been confirmed, deletion requests are send in the fifth and final communication cycle.

Algorithm 3: Pseudo code for the coarsening info method, receiver side.
`coarseningInfoIncoming ( )`

```
receiveQueries ( );
foreach query do
    case task is delete do
        if queryTarget is refined then
            addNewQuery ( queryOrigin, negativeResponse );
            continue;
        else if any queryTarget.neighbours are local and refined then
            addNewQuery ( queryOrigin, negativeResponse );
            continue;
        else if any queryTarget.neighbours are remote then
            addNewQuery ( queryTarget.neighbour, forwardDelete );
        end
    end
    case task is forwardDelete and queryTarget is refined do
        addNewQuery ( queryOrigin, negativeResponse );
    end
    case task is negativeResponse do
        if queryTarget is local then
            removeFromCoarseningCandidates ( queryTarget );
        else if queryTarget is remote then
            addNewQuery ( queryTarget, negativeResponse );
        end
    end
end
```

The coarsening implementation is relatively involved. There are three possible tasks. The `delete` task is issued by grids about to be coarsened to their children. The `forwardDelete` is issued by grids to be deleted to their neighbours, to detect violations of the balance constraints. The third task is called `forwardResponse` and is issued by neighbours of grids marked for deletion in response to the `forwardDelete`. First, the coarsenings are evaluated as far as possible on the origin process. For remote children `delete` tasks are generated, for local children with remote neighbours `forwardDelete` tasks are issued. The procedure on the sender side is again trivial as it simply sends all queries to the receiver.

The method for the receiver side is illustrated in Algorithm 3. A switch case separates the queries according to their task. When the task is `delete`, the target grids are checked if they are refined. If true, a negative response is stored and the next query is treated. If not, all local neighbours of the target grid are checked for refinement. If any of them are refined, a negative response is stored similarly and the next query is treated. Lastly, queries are issued to the remote neighbours with the task `forwardDelete`. For the task `forwardDelete`, the receiver checks if the target grid is refined. If true, a new query is generated with task `negativeResponse`, which will be send either in the same communication cycle when processes swap their sender and receiver state or in the next cycle. The `negativeResponse` task is also the last task differentiated by the switch statement. If detected and if the target grid is local, the grid cannot be coarsened and is removed from the candidates. If it is remote, again a negative response is stored.

After three cycles, all queries are treated and processes have stored a number of negative responses. If the negative responses to coarsening and subsequent deletions are fewer than grids are assigned to this process, additional negative responses are issued, to prevent empty processes. In the fourth cycle, the responses are finally send to the origin processes, where all rejected coarsenings are deleted from the candidate list. Subsequently, the deletion queries are issued in the fifth and last cycle.

Considering the 2:1 tree balance, a refinement can cause consecutive refinements on remote processes, which in turn may prohibit coarsenings. A definitive decision about the validity of a coarsening is consequently only possible after all refinements are evaluated, meaning an interleaving of refinement and coarsening queries is not possible. Therefore, refinements must be evaluated before coarsenings and the amount of communication cycles needed for a complete domain update is given by the total amount of cycles needed for refinement and deletion. Four cycles in two dimensions and six cycles in three dimensions are needed for the information propagation of cascading refinements, plus five cycles for deletion. This amounts to a total of nine (2D), respectively eleven (3D) communication cycles. Adding two cycles for metadata updates, which will be discussed in the next section, the total amount of cycles is eleven (thirteen in 3D).

3.5.3 Neighbourhood Metadata Updates

After the domain updates, new grids, generated through refinement must register their spatial neighbours. Furthermore, already present grids who have gotten new neighbours must register them likewise. The deletion of grids must also be communicated and the affected grids must delete the neighbourhood references accordingly. In the AMR pipeline, the generation of new grids caused by refinements is carried out immediately after all refinement queries have been evaluated. Coarsening grids involves simply to delete children neighbourhood references. This is done after all coarsening queries have been evaluated. The deletion of grids is delayed and carried out after the metadata updates have been completed. Grids about to be deleted have all the neighbourhood references needed to inform the affected grids of the deletion.

When a grid is refined, the task of the update function is to determine the spatial neighbours of the newly generated children grids on the same refinement level. The parent grid is known by construction and second level children grids cannot exist yet because only one additional level of refinement is allowed to be generated per update cycle. Possible neighbour grids are the children of spatial neighbours of the grid that is refined.

Algorithm 4: Pseudo code for the metadata update method, receiver side.

```
metadataUpdateIncoming ( )
  receiveQueries ( );
  foreach query do
      if task is refine then
          if queryTarget is refined then
              addNewQuery ( queryOrigin, positiveResponse );
              addUIDsToQuery (queryTarget.childrenUID);
          else
              addNewQuery ( queryOrigin, negativeResponse );
          end
      else if task is delete then
          removeNeighbourReference ( queryTarget.neighbour, queryOrigin.side );
      end
  end
  sendQueries ( );
```

Algorithm 5: Pseudo code for the metadata update method, sender side.

```
metadataUpdateOutgoing ( )
  sendQueries ( );
  receiveQueries ( );
  foreach query do
      if positiveResponse then
          updateMetadata( queryTarget, neighbourUID );
      end
  end
```

From an implementation perspective, the grid being refined issues a query to its spatial neighbours with the task `refine`. If the refining node, all its spatial neighbours and their respective children are located on the same process, the update can be handled locally. Otherwise all queries to a specific process are send in one message. On the receiver side every query is evaluated first. For a refinement query, the neighbour grid stores a response detailing its refinement status and additionally, if it is refined, it includes the UIDs of its children located towards the query origin. Within the same communication cycle, the receiver sends back its responses to the refinement queries and allows the origin to update its neighbourhood references. This means, after one cycle all new grids caused by a refinement have valid neighbourhood references. On the other side, grids that already existed are not aware of their new neighbours yet. Therefore, in a second communication cycle queries are issued directly targeted at these grids using the already up-to-date neighbourhood references of the refined children. The task for these queries is called `update`.

To update the neighbourhood references of a deletion, grids about to be deleted can simply issue a query to all their spatial neighbours with the task `delete`. Using the `side` information in the query, target grids are able to unambiguously determine the origin and delete its neighbourhood reference. These queries can be exchanged in a single communication cycle.

The order in which individual metadata update queries due to refinements and deletions are processed is arbitrary. Moreover, both operations do not affect each other and may be interleaved. Thus, all update requests are gathered per process and sent in a single message. In conclusion, all metadata updates can be completed in two communication cycles. One for all updates of newly generated grids and all deletions and the second cycle is used to update neighbourhood references of all grids that have gotten new neighbours. The pair of incoming and outgoing methods used in the first update cycle is illustrated in Algorithm 5 and Algorithm 4. The update of neighbourhood references in the second cycle is trivial and therefore omitted.

3.5.4 Complete AMR Pipeline

To conclude the AMR module, the complete program flow of one domain update is illustrated in Algorithm 6. Input for the method are two lists of grids selected for refinement and coarsening (`refinementCandidates` and `coarseningCandidates`).

As explained earlier, refinements and consecutive refinements have to be evaluated before coarsenings. This is done first locally. Afterwards, the remote queries are created and send back and forth four or six times (depending on the problem dimension) using the communication cycle. If refinements caused consecutive refinements, these are added to the `refinementCandidates`. In the next step all `refinementCandidates` are actually refined and new grids are generated.

Afterwards, the `coarseningCandiates` are evaluated. First locally, then the queries for remote processes are gathered and send back and forth three times to allow all queries to reach their intended targets (see section 3.5.2). A fourth cycle is used to send and receive the final responses from remote processes, after which the coarsening can be confirmed or rejected conclusively.

At this point the `coarseningCandiates` only hold valid candidates. Their children are therefore subject to deletion and are gathered in a third list named `deletionCandidates`. Afterwards, the coarsening is carried out. This means the neighbourhood references of all children are deleted, resulting in a negative response for subsequent refinement checks. All remote grids subject to deletion are informed in the following.

Next in line are the metadata updates. After completing all the possible local updates, queries are generated for the remote updates. Two cycles are used to process all queries, with the first cycle combining refinement and coarsening updates and the second cycle is used to forward refinement queries (see section 3.5.3).

Finally, the grids in `deletionCandiates` are deleted and the various candidate lists are cleared. Furthermore, each process needs to update its neighbourhood. Due to refinements, new processes may need to be added to the neighbourhood, conversely, deletions

Algorithm 6: Pseudo code for the AMR workflow

```
Data: refinementCandidates, coarseningCandidates
// Refinement
foreach refinementCandidate do
    evaluateConsecutiveRefinementsLocal ( refinementCandidate );
    getRemoteRefinementQueries ( refinementCandidate );
end
for $i \leftarrow 1$ to $Dimension \times 2$ do
    communicationCycle ( refinementInfoIncoming,
     refinementInfoOutgoing );
end
refineGrids ( refinementCandidates );
// Coarsening
foreach coarseningCandidate do
    evaluateCoarseningsLocal ( coarseningCandidate );
    getRemoteCoarseningQueries ( coarseningCandidate );
end
for $i \leftarrow 1$ to $3$ do
    communicationCycle ( coarseningInfoIncoming,
     coarseningInfoOutgoing );
end
getRemoteCoarseningResponseQueries ( );
communicationCycle ( coarseningResponseIncoming,
 coarseningResponseOutgoing );
// Deletion
getDeletionCandiates ( coarseningCandidates );
coarsenGrids ( coarseningCandidates );
foreach deletionCandiate do
    getRemoteDeletionQueries ( deletionCandiate );
end
communicationCycle ( deletionInfoIncoming, deletionInfoOutgoing );
// Metadata updates
foreach refinementCandidate and coarseningCandidate do
    updateMetadataLocal ( candidate );
    getRemoteUpdateQueries ( candidate );
end
communicationCycle ( metadataUpdateIncoming, metadataUpdateOutgoing );
communicationCycle ( refinementResponseUpdateIncoming,
 refinementResponseUpdateOutgoing );
// clean up
deleteGrids ( deletionCandidates );
clearCandidates ( );
updateNeighbourhood ( );
```

can remove neighbourhood connections between processes. As the communication cycle involves only processes within the local neighbourhood, it is crucial to keep it recent.

3.6 Solution Methods for Systems of Linear Equations

The numerical solution methods were not the main focus of this work. The solution procedure itself is, without one exception, not influenced by the decentralisation of the data structure. For completeness sake, the main points of these methods are detailed in the present section.

In order to discuss the solution methods implemented in the framework, a model problem is analysed. A representative equation is Poisson's equation, an elliptic partial differential equation, which occurs when solving the incompressible Navier-Stokes equations for example. Combining the momentum conservation equation and the mass conservation equation yields an equation for the pressure that takes the form of a Poisson's equation. For a more detailed derivation of equations occurring in Computational Fluid Mechanics see [101].

The Poisson's equation takes the form

$$\begin{aligned} \Delta\varphi &= f && \text{in } \Omega \\ \varphi &= g && \text{on } \Gamma_D \\ \nabla\varphi \cdot n &= h && \text{on } \Gamma_N. \end{aligned} \tag{3.1}$$

Δ is the Laplace operator defined as the divergence of a gradient of a scalar function. As such, Poisson's equation can also be written as

$$\nabla^2\varphi = f \quad \text{or} \quad \nabla \cdot \nabla\varphi = f \quad \text{or} \quad div(grad\varphi) = f. \tag{3.2}$$

φ is the sought solution and f,g and h are given functions. The equation is defined on domain Ω with boundary $\partial\Omega = \Gamma_D \cup \Gamma_N$, whereas Γ_D and Γ_N are the Dirichlet and Neumann boundaries, respectively. Additionally applies $\Gamma_D \cap \Gamma_N = \emptyset$. A Dirichlet boundary condition prescribes the value of the function φ at the boundary directly, a Neumann boundary condition specifies the value of the normal derivative of φ with n denoting the outward directed boundary normal. For simplicity the specialisation of Poisson's equation with right hand side given as $f = 0$, also known as Laplace's equation, is used. For a two dimensional model problem in a rectangular coordinate system, Laplace's equation takes the form

$$\frac{\partial^2\varphi}{\partial x^2} + \frac{\partial^2\varphi}{\partial y^2} = 0. \tag{3.3}$$

Laplace's equation is then discretised using the Finite Difference method on a regular grid similar to the one introduced in Figure 3.1. The values of the continuous function $\varphi(x, y)$ are sampled at $n_i \times n_j$ equidistant spaced grid points, which are subscripted using indices $i = [0, ..., n_{i-1}]$ and $j = [0, ..., n_{j-1}]$.

To approximate the second order derivatives a central difference scheme is used, which takes the form

$$\frac{\partial^2\varphi_i}{\partial x^2} = \frac{\varphi_{i-1} - 2\varphi_i + \varphi_{i+1}}{\Delta x^2} \qquad \frac{\partial^2\varphi_j}{\partial y^2} = \frac{\varphi_{j-1} - 2\varphi_j + \varphi_{j+1}}{\Delta y^2} \tag{3.4}$$

for both x and y-directions. With an equal spacing in both cardinal directions $\Delta x = \Delta y$, substituting the derivatives in equation 3.4 into Laplace's equation 3.3 yields

$$\varphi_{i-1,j} + \varphi_{i,j-1} - 4\varphi_{i,j} + \varphi_{i+1,j} + \varphi_{i,j+1} = 0. \tag{3.5}$$

This is the general equation for each internal point in the grid. Values at the boundary of the domain are set up accordingly from the equations given at the boundary (cf.3.1). In conclusion, one ends up with an equation for the value at every grid point, in other words a system of coupled linear equations. This system can be assembled into the form

$$Ax = b \tag{3.6}$$

with A being the coefficient matrix, x the vector of unknowns, in this case the values of φ at the grid points and b being the right hand side of the equation.

Standard procedures to solve systems of linear equations are direct and iterative methods. Among the direct methods are the LU decomposition based on classical Gaussian elimination or the Cholesky decomposition, which requires less operations but only works for a symmetric positive definite matrix A. Another direct method is the QR decomposition which in itself can be used to compute a solution, however, it is also used as part of many more involved iterative methods to find an orthonormal basis. Direct methods compute the solution within one pass of the respective algorithm. Unfortunately, these methods also entail large memory requirements and comparably large runtime complexities, which limits their applicability for large systems.

Here, iterative methods are used, which compute increasingly accurate approximations to the solution with every pass of the respective method. As each pass is relatively cheap compared to direct methods, iterative methods are usually able to produce reasonably good approximations of the solution with a limited number of iterations, resulting in a much lower overall time and, depending on the method used, lower memory requirement. Among the iterative methods are splitting methods, like Jacobi and Gauss-Seidel with or without weighting, Krylov subspace methods, such as the conjugate gradient and GMRES (Generalized Minimal Residual) and finally, multigrid methods which combine various of the aforementioned methods on a hierarchy of discretisations. For a conclusive overview over these methods to solve systems of linear equations, the interested reader is referred to [72] and [210] (in German).

Implemented in the framework are the weighted versions of Jacobi and Gauss-Seidel methods. These methods in general perform well in reducing short wavelength errors, but long wavelength errors are reduced very slowly, leading to slow convergence rates. To accelerate the time to solution, a multigrid method is implemented as well, which is adapted to and makes use of the hierarchical data structure. As explained in the beginning, the idea for these methods and especially their adaption for the present hierarchic data structure is taken from the work of Frisch [113].

Additionally, an asynchronous version of the parallel Jacobi method has been implemented. An asynchronous method is characterised by a parallel iterative method, where the individual processes do not wait for neighbours to complete their current iteration step and use the latest values available. It is possible for these methods to converge faster or even at all, compared with a synchronous method, because processes do not wait idle for other processes. In view of the novel repartitioning strategy, which gives a slightly worse load balance compared to global methods, asynchronous methods are a way to alleviate these imbalances and use otherwise idle time in which less burdened processes wait for results from their neighbours.

In the following, an overview over the implemented methods is given.

3.6.1 Splitting Methods

As their name suggests, these methods are based on the splitting of the matrix A into

$$A = B + (A - B). \tag{3.7}$$

Applying this split for the operator A in the system of linear equations 3.6 yields

$$Bx + (A - B)x = b \qquad \equiv \qquad Bx = (B - A)x + b. \tag{3.8}$$

If B is a regular matrix, one can invert it and ends up with

$$x = B^{-1}(B - A)x + B^{-1}b. \tag{3.9}$$

From here, one is able to define the linear iteration method with $M = B^{-1}(B - A)$ and $N = B^{-1}$

$$x^{m+1} = Mx^m + Nb \quad \text{ for } m = 0, 1, ... \tag{3.10}$$

Here, m denotes the current iteration count. Starting with an initial guess of the vector of unknowns x^0, one supplies the current approximation to the solution x^m into the right hand side equation which yields a new approximation x^{m+1}. It can be shown that this method with iteration matrix M is consistent and will convergence to the exact solution given an infinite number of iterations if and only if the spectral radius of the iteration matrix $\rho(M) < 1$ (see [210]).

The question now becomes how to chose the split of matrix A. Solving the original system of linear equations $Ax = b$ directly is done by computing the inverse of A and applying it to both sides of the equations. That is

$$x = A^{-1}b. \tag{3.11}$$

Comparing above equation 3.11 and the iteration method 3.9, one observes that B can be viewed simply as an approximate to the matrix A and the approximation error is successively corrected with $B^{-1}(B-A)x_m$. A perfect approximation $A = B$ consequently yields equation 3.11, allowing the solution of x in one step with no correction. The problem here is, finding the inverse of A is computationally very expensive and makes direct methods infeasible for large matrices. Therefore, an iterative method tries to find a good approximation B of the original matrix A that is easily invertible.

Popular choices for B are as follows:

The Richardson method
$B = \frac{1}{\omega}I$ with $\omega \neq 0$ being a weighting factor and I is the identity matrix.

The Jacobi method
$B = D$ with $D = diag((a_{ii})_i)$ in other words the diagonal elements of A.

The weighted Jacobi method.
$B = \frac{1}{\omega}D$. A version of the Jacobi method with a weighting factor $\omega \neq 0$.

The Gauss-Seidel method
$B = D + L$ and L is the strict lower diagonal part of A.

The successive over-relaxation method (SOR)
$B = \frac{1}{\omega}D + L$. A weighted version of the Gauss-Seidel method.

3.6.1.1 Jacobi Method

Using $B = D$ in equation 3.10 and after some rearranging yields

$$x^{m+1} = D^{-1}(b - (A - D)x^m). \tag{3.12}$$

The inverse of a diagonal matrix D is the reciprocals of its diagonal elements $D^{-1} = diag((\frac{1}{a_{ii}})_i)$. The expression $(A - D)$ simply is the matrix A with zeros on all diagonal elements. Therefore the component-based notation for every unknown x_i^{m+1} can be expressed as

$$x_i^{m+1} = \frac{1}{a_{ii}} \left(b_i - \sum_{\substack{j=1 \\ j \neq i}}^{n} a_{ij} x_j^m \right). \tag{3.13}$$

One observes that every new approximate x^{m+1} is computed solely using the old approximate x^m. Therefore, the Jacobi method is also called total-step method. When implementing the method, two vectors are needed to store the new and old values x^{m+1} and x^m, as to not overwrite x^m within the current iteration.

Using $B = \frac{1}{\omega}D$ in equation 3.10 and after some rearranging one ends up with

$$x^{m+1} = x^m + \omega D^{-1}\left(b - Ax^m\right). \tag{3.14}$$

Thus x^{m+1} can be interpreted as the current approximation x^m plus some correction $D^{-1}\left(b - Ax^m\right)$. Using an additional weighting ω on the correction term, the aim is to accelerate convergence. This can be achieved by minimising the largest eigenvalue of the iteration matrix M of the method and consequently decrease the spectral radius $\rho(M)$ to speed up the convergence.

The corresponding component-based notation of the weighted Jacobi method is

$$x_i^{m+1} = (1-\omega)x_i^m + \frac{\omega}{a_{ii}}\left(b_i - \sum_{\substack{i=1\\ j\neq i}}^{n} a_{ij}x_j^m\right). \tag{3.15}$$

3.6.1.2 Gauss-Seidel Method

Substituting $B = D + L$ into equation 3.10 and with $U = A - D - L$, the strict upper diagonal part of A, yields after some rearranging

$$x^{m+1} = (D+L)^{-1}(b - Ux^m). \tag{3.16}$$

Computing the inverse of $(D+L)$ is no longer trivial, however using forward substitution, the components of x^{m+1} can be computed sequentially. In component-based notation the equation for every unknown x_i^{m+1} is

$$x_i^{m+1} = \frac{1}{a_{ii}}\left(b_i - \sum_{j=1}^{i-1} a_{ij}x_j^{m+1} - \sum_{j=i+1}^{n} a_{ij}x_j^m\right). \tag{3.17}$$

In this notation, the Gauss-Seidel method is almost identical to the Jacobi method. The main difference is that already computed new components of x^{m+1} are used within the current iteration. As such, only one vector of unknowns can be used to implement the method. Newly computed values directly overwrite the old ones. The spectral radius of the iteration matrix M is generally smaller compared to the Jacobi method, leading to a faster convergence. However, because the computation of specific components of x^{m+1} requires already computed values in the current iteration, dependent on the matrix structure A, the critical path to compute these components might be very long. These dependencies prohibit an efficient parallelisation of the method.

Similar to the weighted Jacobi method, a weighted version of the Gauss-Seidel method can also be conceived. This is known as Successive Over-Relaxation method (SOR). Substituting $B = \frac{1}{\omega}D + L.$ into equation 3.10 yields

$$x^{m+1} = (D+\omega L)^{-1}(\omega b - [\omega U + (\omega - 1)D]x^m). \tag{3.18}$$

Using forward substitution again, yields the component based notation

$$x_i^{m+1,} = (1-\omega)x_i^m + \frac{\omega}{a_{ii}}\left(b_i - \sum_{j=1}^{i-1} a_{ij}x_j^{m+1} - \sum_{j=i+1}^{n} a_{ij}x_j^m\right). \tag{3.19}$$

3.6.1.3 Matrix-Free Approach

In general, after the assembly of A, one of the methods described above can be applied to solve for the unknown quantities x. However in the special case of the discretised Laplace's equation 3.5 the assembled matrix without boundary conditions has only five non-zero entries per row in two dimensions and seven non-zero entries in three. Furthermore, all entries in matrix A are consistent for each internal point. That means, the calculation of the value of each point uses the same coefficients. Applying the Jacobi method 3.13, a new approximate solution of $\varphi_{i,j}^{m+1}$ is evaluated as

$$\varphi_{i,j}^{m+1} = -\frac{1}{4}\left(\varphi_{i-1,j}^{m} + \varphi_{i,j-1}^{m} + \varphi_{i+1,j}^{m} + \varphi_{i,j+1}^{m}\right). \tag{3.20}$$

Applying the Gauss-Seidel method 3.17 looks almost identical, except the already computed values of the current iteration are used directly:

$$\varphi_{i,j,}^{m+1} = -\frac{1}{4}\left(\varphi_{i-1,j}^{m+1} + \varphi_{i,j-1}^{m+1} + \varphi_{i+1,j}^{m} + \varphi_{i,j+1}^{m}\right). \tag{3.21}$$

For the solution of Laplace's equation with the given discretisation, it makes little sense to assemble the matrix, as the structure is consistent and sparse for each internal point. As such, the solution procedure simply solves the above equation for each internal point. When boundary conditions are supplied, the equation has to be altered. For a mesh point with a Dirichlet condition for example, no equation has to be solved at all because the value is prescribed. As the matrix A is never assembled, solving the equations in this fashion is called matrix-free.

The drawback of this matrix-free approach is that it is very rigid and only solves the exact equation with the supplied discretisation. If the equation or the discretisation changes, a new solver has to be implemented. For example, a higher order discretisation, involving more data points, leads to a different matrix A and the equations have to be altered accordingly.

3.6.1.4 Parallelisation

When it comes to parallelisation, the Jacobi method is straight forward. Every process computes a full sweep over all its assigned grids. Meaning it evaluates a new approximate solution for every grid point using only the old values. After the sweep all ghost cells are updated and a new iteration is computed.

The Gauss-Seidel method is harder to parallelise due to the dependencies within a single iteration. One remedy is the red-black version of the Gauss-Seidel method that can be applied for specific discretisations. Here, the components of x are split into two non-overlapping sets with no direct dependencies within the two sets. The left illustration in Figure 3.29 shows the partitioning of the grid points for the discretised two dimensional Laplace problem into two sets. Now, within each set, new approximations for x can be computed completely in parallel. Each process runs one sweep over one of the sets, updates the respective ghost cells and repeats the process with the other set until convergence is

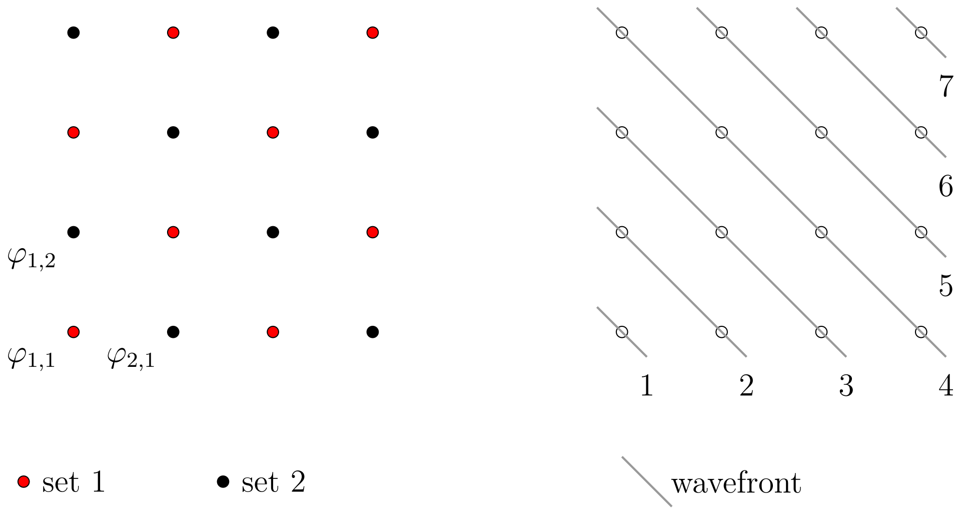


Figure 3.29: Partitioning of grid points to parallelise the Gauss-Seidel method for a two dimensional Laplace problem. Red-black partitioning on the left. Wavefront partitioning on the right

reached. Even though this is a possible remedy to parallelise the Gauss-Seidel method, one should be aware that it alters the original iteration procedure as even though the sets have no direct dependencies within them, they are implicitly coupled via the second set. This dependency is not satisfied anymore.

A second possible way to parallelise the method is a wavefront approach. Again, given the two dimensional discretisation of Laplace's equation and a Dirichlet boundary condition on the value $\varphi_{1,1}$, all dependencies for the computation of $\varphi_{2,1}$ and $\varphi_{1,2}$ are satisfied. They can be computed in parallel. After both have been computed, all dependencies for the computation of $\varphi_{3,1}$, $\varphi_{2,2}$ and $\varphi_{1,3}$ are satisfied, which again can be evaluated in parallel. The right illustration in Figure 3.29 shows the partitioning of the grid points into wavefronts. Within each wavefront, the grid points have no dependencies and can be computed completely in parallel. However, wavefronts are dependent on their predecessor, meaning new approximations for the grid point values in wavefront one must be completed before grid point values in wavefront two can be evaluated. The wavefront approach works for a partitioning of grid points within a grid and it also can be applied on a complete grid level. The drawback of a wavefront partitioning is the relatively high idle time where processes need to wait for the resolution of dependencies. Given the example in Figure 3.29 with 4×4 grid points and four available processes, while one process computes an approximate for $\varphi_{1,1}$ the other three are idle. In wavefront two, two processes can work in parallel while the other two are idle. Only in wavefront four, all processes could be utilised, before more and more processes again become idle in later wavefronts. Furthermore, there can never be more participating processes as there are grid points (or grids) in the largest wavefront.

Implemented in the framework are both the weighted Jacobi method as well as the SOR for computing sweeps on individual grids. That means, the Jacobi method can be used to iterate over the complete domain in parallel with regular ghost cell updates after each sweep. For the SOR method this is not as easily possible due to the discussed dependencies in its formulation. A red-black or wavefront partitioning for parallelising the method is not used. The benefits of introducing a multigrid method, which involves both solvers, outweighs those benefits gained by a parallel version of the SOR.

The solution procedure itself is not affected by the decentralisation of the data structure. All ghost cell exchanges happen between neighbouring grids and consequently between neighbouring processes without any global data synchronisation. Though, in Frisch' implementation, a global sum of the residual was computed after every sweep to determine whether the solution has converged and the iteration process could be stopped. As the Jacobi method is never used on its own until the solution has converged this global communication never takes place. Further considerations concerning the stopping criterion and their influence on the communication patterns are discussed in the following sections for the multigrid method and the asynchronous methods.

3.6.2 Multigrid Method

Doing a Fourier-analysis on the initial guess for x to separate different wavelength components of the error, one finds the eigenvectors associated with eigenvalues of A are simply the Fourier modes of the error. Furthermore, the smoothest mode, meaning the error component with the lowest frequency is associated with the highest eigenvalue and vice versa. Even with a weighting factor, the iterative methods discussed above are not able to reduce the largest eigenvalue considerably. In practice this means that they are able to reduce high frequency errors quickly, but take many iterations to reduce smooth, low frequency errors of the initial guess. Consequently, these methods are also termed relaxation or smoothing methods.

However, a second observation can be made. Smooth quantities on fine grids can be transferred to coarser grids, where they behave more oscillatory. In combination these principles have led to the development of multigrid methods. Multigrid methods have been first proposed by Brandt in 1977 [33]. They have since then successfully been applied to a variety of applications, including hyperbolic differential equations such as the Euler equations [117] and hyperbolic-parabolic differential equations such as the Navier-Stokes equations [7]. The interested reader is referred to [305], [35] and [132] for a more comprehensive overview over multigrid methods. Convergence properties of the method can be found in [131] or [132] for example.

In the following, an abbreviated overview over the method is given. The observations made, concern the error rather than the solution of the system of linear equations. Therefore, $Ax = b$ must be rewritten in terms of the error first. By introducing the residual r, which is defined as

$$r = b - A\tilde{x} \tag{3.22}$$

where $\tilde{x}$ is an approximate to the exact solution $\hat{x}$, and the error as difference between the exact solution and the approximate

$$e = \hat{x} - \tilde{x}, \tag{3.23}$$

one is able to write an equivalent system

$$Ae = r = b - A\tilde{x}. \tag{3.24}$$

This means, instead of solving for the exact solution, it is equivalent to solve for the error and use it to correct an approximate solution.

To complete all necessary preliminaries for a multigrid method, one needs a way to transfer quantities from fine grids (Ω^h) to coarse grids ($\Omega^{2h}, \Omega^{4h}...$) and vice versa from coarse grids to fine grids. These operations are called restriction, i.e. the downsampling of a quantity from a fine grid to a coarse grid, and prolongation, i.e. interpolation of a quantity from a coarse grid to a finer grid. Most common restriction operators are injection and full weighting operators. The most common prolongation operator is the linear prolongation.

The injection operator to restrict quantities is comparably simple. To get the coarse grid representation, the quantity discretised at every nth point is taken, without considering the left out grids points. For example, considering only every second point leads to a coarser grid with about half as many grid points in each coordinate direction. The full weighting operator also takes into account a restriction to every nth grid point. However, contributions from left out grid points are evenly distributed among their remaining neighbouring grid points. The linear prolongation functions are exactly contrary to the full weighting. The quantities at each coarse grid point are interpolated to their fine grid counterpart as well as to neighbouring fine grid points who do not exist on the coarse grid. Choosing full weighting and linear prolongation as restriction and prolongation operators, one observes that one operator is simply the transpose times a constant factor of the other. This is called variational property. A combination of two operators who exhibit this property is desired, because it signifies a well-defined way of transferring quantities between grids without discrepancies between the operators.

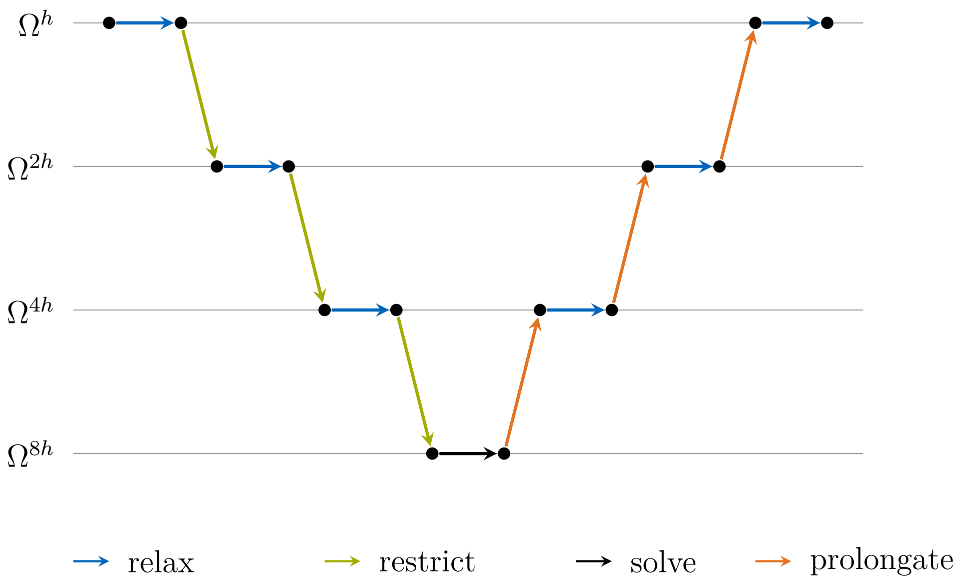


Figure 3.30: Multigrid v-cycle scheme.

Algorithm 7: Recursive multigrid v-cycle scheme.

```
procedure x ← MGv (x,b)
// Pre-relaxation
for $i \leftarrow 0$ to $\alpha_1$ do
    x ← relax (x,b);
end
// compute residual
r ← b − A x;
// restriction
r ← restrict (r);
// recursive call until the coarsest grid
if ($\Omega^l > \Omega^{min}$) then
    e ← MGv (e,r);
else
    // solve on the coarsest grid
    e ← solve (e,r) ;
end
// prolongation
e ← prolongate (e);
// correction
x ← x + e;
// Post-relaxation
for $i \leftarrow 0$ to $\alpha_2$ do
    x ← relax (x,b) ;
end
```

Combining all the components into a recursive solution scheme is illustrated in Algorithm 7. First, α_1 iterations of a relaxation method are performed on the finest grid to reduce high frequency errors. In the second step, the residual is computed and restricted to the next coarser grid. As long as the method has not reached the coarsest level, the method is recursively called to compute the error from the current residual. If the coarsest grid has been reached, the number of remaining grid points is usually sufficiently small to apply a direct method to solve for the error. The error is then prolongated to the next finer grid and used to correct the current solution. Finally, α_2 iterations of a relaxation method are performed on the corrected solution.

In Figure 3.30 the flow of a single iteration of the algorithm is illustrated for the initial resolution Ω and three coarser resolutions Ω^{2h}, Ω^{4h} and Ω^{8h}. The structure of the recursive function leads to the characteristic v-cycle shape of the method. However, other formulations lead to other characteristics shapes. Alternatively, the recursive function call may be executed multiple times, which leads to the w-cycle. The w-cycle takes more effort per iteration as it spends more time on coarser levels, leading to better convergence properties over the v-cycle. The second alternative is the f-cycle. This variation starts by building up a good initial guess through nested iteration, before running the standard v-cycle. Again, this shows superior convergence over the v-cycle, at the cost of a higher computational effort per iteration. The choice which cycle has the best properties is usually problem dependent.

3.6.2.1 Multigrid-Like Method

By comparing the bottom-up and top-down communication stages to synchronise the data structure to the restriction and prolongation operations of a multigrid method, Frisch emphasises the similarity between them. The bottom-up stage aggregates data on each level and recursively sends it upwards to the respective coarser levels until the root grid, i.e. the coarsest representation, is reached. Conversely, the top-down stage runs from the coarsest grids to the most refined ones, interpolating data into the respective cells on each intermediate level. As such, the structure naturally lends itself for the application of a multigrid method, using the communication stages as restriction and prolongation. The advantage here is that coarser grids do not have to be extracted from finer grids as they are already present within the data structure.

The easily parallelisable weighted Jacobi iteration is used as relaxation method on all levels except the coarsest, which is represented by the single root grid. Here, due to its superior convergence properties, the SOR method is used to solve for the error. As the solution is conducted only on a single grid on a single process, parallelisation is not an issue.

Upon closer inspection, the grid data exchange procedures closely resemble the restriction and prolongation operators, however, they do not fit perfectly. The main difference is that the location of kept grid cells in coarser representations matches that of finer representations in the classical multigrid formulation. Depending on the subdivision chosen, this is not the case in the grid hierarchy used in the framework. Therefore, an interpolation from different grid point locations across resolutions is necessary. Due to this discrepancy, the method is termed multigrid-like.

For a conclusive description of the method, its intricacies, convergence studies and benchmark results, the interested reader is once again referred to the work of Frisch [113]. One minor difference between the implementation of Frisch and the one used in the decentral framework is the convergence detection. In the former, after every full iteration, a global sum over the Euclidian norm of the residual vector was computed to detect convergence. From equation 3.24 it can be concluded, when the residual goes to zero, the error does as well. Meaning the residual is an equally valid measure to detect the convergence of the solution. The decentral framework tries to limit global operations as far as possible. Therefore, to accumulate the residual, the norm is computed on the finest grids and send in conjunction with the regular data during the bottom-up communication towards the root grid. Intermediate results are accumulated on the intermediate levels and forwarded to the next coarser level. The global residual norm is then available on the root grid. When the norm is sufficiently small, the process holding the root grid sends a halt message in conjunction to the data with the top-down communication stage.

3.6.3 Asynchronous Methods

Up to now, when the iterative methods described above are parallelised, a synchronous approach was used. This means, there is a clear order of execution of the different parts of the solution procedure. The simulation domain is decomposed into non-overlapping regions (the grids), which are assigned among the participating processes. Additionally, each grid is surrounded by a ghost layer, which represents the current quantities of neighbouring regions, which are needed to update the local solution. During the solution process, each process executes an update sweep over all its assigned grids and afterwards, the corresponding ghost cells are updated with the latest values before the next iteration can commence. In other words, each grid sweep always uses values from the same iteration to update the local values. Consequently, all processes are synchronised within the current iteration they compute.

This behaviour matches exactly with a serial implementation of an iterative method, where only one large grid on a single process would be used. In general, the parallelisation should not influence the solution in any way, other than speeding up the computation. The drawback however, is when load is unevenly distributed. In the present case, when some processes are burdened with a much higher amount of grids than others. Because of the necessary synchronisation after every sweep, less burdened processes wait idle until other, more burdened processes catch up and allow the exchange of the latest computed values to carry on with the next sweep. The most obvious remedy is an efficient repartitioning, i.e. a load-balancing, which will be discussed in detail in the next chapter.

Another possible remedy to alleviate load imbalances are asynchronous iterative methods. Here, every process simply uses the latest available neighbouring quantities without synchronisation. More explicitly, every process executes a sweep over all its assigned grids, as before. However, instead of waiting for neighbouring processes, a quick check is executed whether any previous communication has been completed. If a previous outgoing message to transfer ghost cell values has been completed, a new message with the latest values is issued. Likewise, if new neighbouring cell values have arrived, they are extracted into the local ghost layers and are used in the next iteration.

For a better understanding, Figure 3.31 illustrates the iteration process for a synchronous and an asynchronous iterative method. The simulation domain is split into two grids which are distributed to two processes p 0 and p 1. Each grid has a layer of ghost cells representing the neighbouring values on the other grid. Furthermore, calculating a new approximate solution x_{m+1} for each grid point using some iterative approach takes twice as long on process one compared to process two. In the synchronous case, process zero finishes the first iteration, sends boundary values to update the respective ghost cells and has to wait for process one to send the newest values of its current iteration, before it can commence with the next iteration. In the synchronous case, all processes work on the same iteration with the latest values. The method does not differ in any way from its non-parallelised version. In the asynchronous case, process zero finishes its first iteration, sends the respective boundary values and immediately starts with the second iteration without waiting for updated ghost cell data. In this case, iteration one and two use the same ghost cell values from the same iteration on process one. After the second iteration, process zero does not send data of the newly computed approximate because the last message has not been received yet. This is however implementation dependent, as one could also have multiple messages in flight. In the asynchronous case, different processes can be on different iterations. Additionally, the parallelised version differs considerably from its single process counterpart. Furthermore, depending on the load situation in the machine, different runs of an asynchronous method may differ from each other.

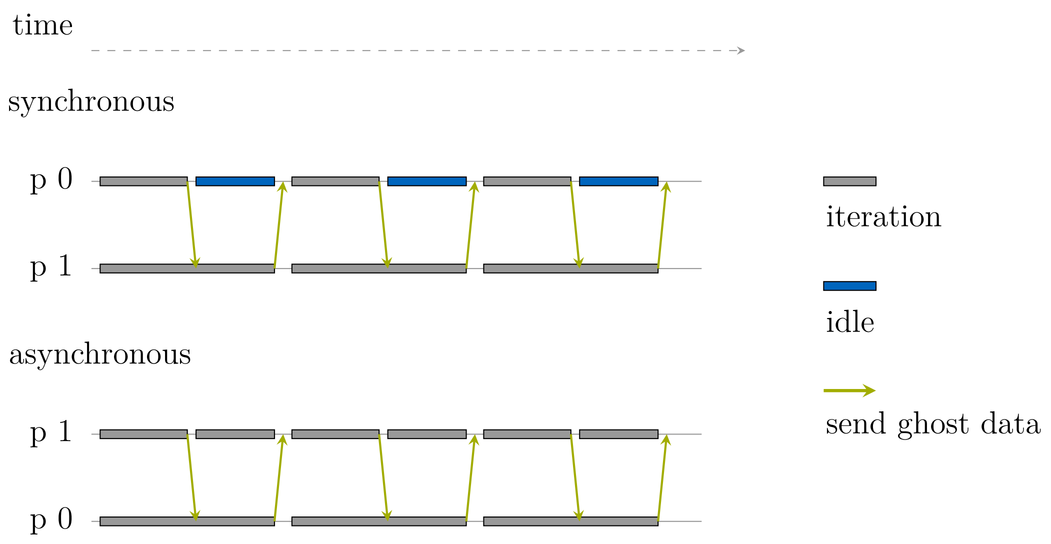


Figure 3.31: Comparison between synchronous and asynchronous iteration methods

It is obvious that asynchronous methods alter the behaviour and the convergence properties of the underlying method considerably. Chazan and Miranker were the first ones to explore such asynchronous methods in 1969 [54]. Under the term chaotic relaxation, they tried to establish a mathematical framework and derive convergence guarantees. Baudet later extended this framework, giving actual measurements on multiprocessor systems [19]. For synchronous iterative methods, their convergence has been established if and only if their iteration matrices have a spectral radius $\rho(M) < 1$. For asynchronous iterative methods this condition is aggravated by requiring the spectral radius of the component-wise absolute value of the iteration matrix $\rho(|M)| < 1$. If $\rho(|M)| \geq 1$, then there exists a sequence of asynchronous iterations, such that the method does not converge. However, depending on the initial guess and the sequence of asynchronous iterations, which are influenced by the current load situation as well as the used hardware, the asynchronous method might still convergence. Even in cases where the synchronous method does not [332, 334]. As no guarantees can be given in these cases, this poses a rather theoretical benefit.

3.6.3.1 Asynchronous Jacobi Method

Corresponding to the idea of parallelising an iterative procedure with asynchronous ghost cell updates, an asynchronous Jacobi method has been implemented in the framework.

Given a uniform domain refinement, the crucial points of the implementation are the asynchronous domain updates as well as the convergence detection. The later are explained in more detail in the next section. If an adaptive domain refinement is given, it has to be determined which grids actually need to exchange data first (see also section 3.3.1.4).

The sweeps to update the local quantities work exactly the same as for the regular Jacobi method. However after each sweep, the asynchronous operations to update ghost cells take place. These can be implemented using regular MPI peer-to-peer communication which requires explicit send and receive calls from both participating processes. Another recent alternative is the use of MPI's one-sided functionality, also known as remote memory access (RMA), which allows writing or reading data to another processes' memory without the other processes involvement [338]. The current implementation uses the former, "classical" peer-to-peer MPI facilities.

After determining the communication structure, every grid posts a non-blocking `MPI_Irecv` for every neighbour, spatial and hierarchical with which it needs to exchange data, to receive updated ghost cell data. After the initial sweep, a non-blocking `MPI_Isend` is posted, to send new ghost cell data to the applicable neighbours.

Algorithm 8: Asynchronous iteration method

```
// sweep
x ← relax (x,b)
// asynchronous receive and send
foreach neighbour do
    if MPI_Test (recvRequest) = complete then
        extract (buffer.neighbour);
        MPI_Irecv (buffer.neighbour, recvRequest);
    if MPI_Test (sendRequest) = complete then
        MPI_Isend (cellData.neighbour, sendRequest);
end
```

Non-blocking MPI calls are supplied with an additional request object over the blocking variants. This status object allows to check or wait for the completion of the send or receive call. After every regular sweep, the completion of the `MPI_Irecv` call is checked using the `MPI_Test` method. If the message has been received, the receive buffers are extracted into the local ghost cells and a new `MPI_Irecv` for the next update is posted. The commencing sweep will use the newest updated ghost cell values. If nothing has been received so far, execution of the next sweep is not halted, instead, the next sweep simply uses old ghost cell values to update the local quantities. Similarly, after each sweep the completion of the `MPI_Isend` call is checked using the `MPI_Test` method and the corresponding request object. If it has been completed, a new `MPI_Isend` call is issued with the latest neighbouring values. If not, the method simply continues with the next sweep. In algorithm 8, the algorithm is illustrated in a simplified fashion.

As mentioned before, in a uniform refinement setting, all leaf grids simply execute the asynchronous iteration algorithm until convergence. Coarser grids do not have to be involved. In an adaptive setting, it is sensible to determine which grids actually need to be involved in actual communication, prior to the asynchronous solution procedure. In addition to the horizontal communication between leaf grids, whenever a leaf grid has no valid spatial neighbour at one of its sides, the ghost layer exchange must involve hierarchical communication via its parent. Leaf grids are able to determine this case completely local. They simply check whether any of their spatial neighbourhood references is invalid. If that is the case, the ghost layer transfer in the respective direction will be executed via the parent grid. Non-leaf grids cannot determine their status locally, because they have no information whether their spatial neighbours are refined or not. Therefore, this refinement status has to be exchanged between all spatial neighbours once at the beginning of the solution process. More specifically, if no change to the domain happens in between timesteps of the simulation, the communication structure remains valid.

The exchange of refinement status is implemented using a non-blocking `MPI_Irecv` call to receive the status of every neighbour, followed by a blocking `MPI_Send` of a grids' own status and a subsequent `MPI_Waitall` to block further execution until all messages have been received.

Now, all non-leaf grids are able to determine whether they are involved in transferring ghost layer data. This is the case for a non-leaf grid with leaf spatial neighbours. Here, the grid's children on the side of the leaf neighbour have no spatial neighbours on their level. They communicate their respective ghost cell data to the parent which forwards it and vice versa. In asynchronous fashion, data is extracted to the local data cell whenever new messages have been received and forwarded if a previous message was sent successfully.

3.6.3.2 Convergence Detection

The stopping criteria in asynchronous iterative methods is not trivial and has been studied for example by Bahi et. al. for decentral applications [16, 15]. In the synchronous case, the solution process is halted either when the norm of the global residual is small enough, or after crossing a threshold of the number of global iterations. The later cannot be applied at all for the asynchronous case, since there is no global synchronous iteration every process is working on.

The algorithm described by Bahi is based on local convergence. Every grid individually tracks whether their local residual norm is below a defined threshold. If this is the case, a termination request is send to a central process tasked with evaluating the termination. After every grid has send the termination request to the central instance, a verification response is sent in response. It might happen that after the first termination request sent by a grid, the solution is not within the local convergence threshold any more. This is the case, when delayed ghost cell updates cause large discrepancies with a neighbouring grid's cell values. If a residual norm of a grid is still in a valid range when receiving the verification request, it sends a confirmation. Otherwise it sends a negative response to the central process. If all grids have sent a confirmation, the solution procedure can be halted and the central instance sends final termination requests to all processes. Otherwise, a negative query is sent to all processes and the convergence detection is started anew. In Figure 3.32 the mechanism to detect the convergence is illustrated for four processes, whereby process zero acts as detection instance. In this example, all processes are still within the local convergence threshold when the confirmation request arrives. Consequently, convergence is confirmed and the solution process is terminated.

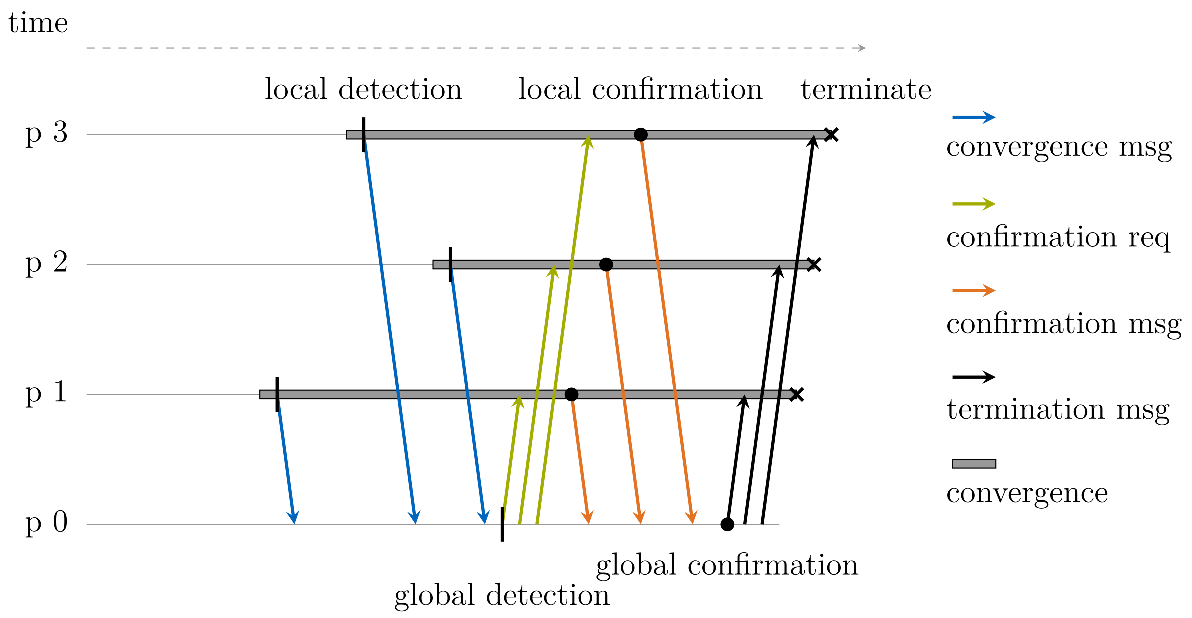


Figure 3.32: Global convergence detection process. Adapted from [15].

This method is also implemented in the present framework, however in a simplified fashion without confirmation of the convergence. Whenever a grid has reached local convergence, it sends a termination request to its parent grid. Here all requests are accumulated and if all have arrived, a termination request is send to its parent in turn. As soon as the root grid has received all termination requests from its children, a halt signal is send in the opposite direction, down the tree. In the mean time all leaf grids and grids needed to forward ghost cell data continue iterating until they have received the halt order. Finally, all outstanding messages need to be received and no new receive and send calls must be issued. This is achieved by appending the last outgoing message from each grid that has received the halt order with a termination flag, signifying this to be the last communication from this grid.

An alternative method implemented, is via the evaluation of the norm of the global residual. Instead of sending a single termination request, the local residual norm of each leaf grid is send continuously in an asynchronous fashion towards the root grid. The individual children norms are accumulated on each intermediate level and the result is forwarded. The root grid can then determine global convergence and send the corresponding halt orders.

3.6.3.3 Conclusions and Outlook for Asynchronous Methods

There are a few key issues when discussing the applicability of asynchronous methods in practice. When using a local convergence criterion, the overall solution can never be worse than when using a synchronous method. The overall simulation time is determined by the slowest process. It needs to iterate until it has reached the local convergence threshold. Instead of waiting idle, faster processes use the time to carry out more iterations towards the exact solution and the norm of the global residual is lower compared to a synchronous method. This is true even when using old ghost cell values [332]. It may even happen that the asynchronous method is faster. This is because the grids on the slowest process may converge faster due to more exact values from their neighbours assigned to faster processes.

Using a global convergence criterion is generally ill-advised. Firstly, the frequent accumulation of the norm of the global residual entails much more communication compared to the single termination request of the former method. Secondly, even though the computation halts at the same norm as the synchronous method would, the residual may be badly distributed. High local residuals in grids on slow processes can be balanced by low local residuals in grids on faster processes, masking a non-converged local solution.

In conclusion, an asynchronous method with a local convergence criterion can be beneficial when there is a considerable load imbalance or the hardware is heterogeneous. Given the fact, one of the main goals of the present work is to introduce an efficient dynamic partitioning to alleviate these imbalances, the applicability is limited. However, the iteration count per grid could be used as a more accurate load metric, compared to simply taking the number of grids per process. Therefore, asynchronous methods still could be a valuable addition to the modules of the decentral framework.

Currently, the only asynchronous method implemented in the framework is the asynchronous Jacobi method described above. This method suffers from the same poor convergence properties stemming from the slow reduction of short wavelength errors. As such, it cannot compete with the synchronous multigrid method. Recently however, asynchronous versions of the multigrid method have gotten some attention by Wolfson-Pou [331] and Chow et. al. [333]. They base their work on original ideas of McCormick et. al. from 1989 under the term of asynchronous multilevel adaptive methods (AMA) [138, 208], which were later advanced and termed asynchronous fast adaptive composite-grid methods (AFAC) [194, 195]. A robust asynchronous multigrid method could be a valuable extension to the decentral framework in the future.

3.7 I/O

To conclude all modules affected by the decentral structure, a brief overview of the I/O functionality of the framework is given. For visualisation and checkpointing, the framework uses the Hierarchical Data Format version 5 (HDF5). Initially developed in 1987 by the National Center for Supercomputing Application (NCSA), HDF5 was selected by NASA to be used in its Earth Observing System [189].

HDF5 was developed to efficiently store large amounts of array based data. To this end, HDF5 provides an abstract data model to facilitate the view of the stored data. The structure within an HDF5 file is made up from groups, the data is held by datasets. A group may contain datasets and additional groups, resulting in a hierarchical tree-like structure. Additionally, HDF5 provides functionality for distributed memory systems. Parallel HDF5 routines are based on MPI-IO, whereas the HDF5 libraries manage the application's I/O calls and in turn utilize MPI-IO's routines, providing easy-to-use parallel I/O functionality.

As mentioned above, the I/O functionality involves two main tasks. Writing checkpoint files with all necessary data from which a simulation can be restarted and files with pure visualisation data. The structure of a checkpoint file is as follows. On the top level, there exist two groups, one for persistent simulation data, for example the refinement specification, the current time and the time step etc. The second group contains the current grid data. That includes metadata and simulation data. The structure of a visualisation file contains one group for every written time step. In the simplest case, only the most refined grid data for every spatial point is written such that the simulation domain is represented in the finest resolution available. Another possibility is to store the data of every grid in addition to the hierarchical neighbourhood structure of every grid. This allows to traverse the hierarchical refinement structure starting from the root grid and allows a selective visualisation of parts of the simulation domain in various resolutions. The advantage is the ability to limit the amount of data to be visualised, allowing a fast overview over coarse features or a more detailed view of selected parts of the domain (see also [229, 93, 94]).

The parallel functionality of HDF5 allows all processes to access a single shared file independently. Each process is assigned an exclusive region inside the file for read and write operations. There are however, two collective communication calls necessary when generating and writing this single shared file. The generation of the file as well as the generation of the file structure is a collective operation that needs to be carried out by all processes. Here, the total size of the domain must be known. MPI provides the `MPI_Allreduce` method, a global reduction operation that allows to compute the sum of all reduced values when supplied with the `MPI_SUM` method. This is used to compute the total number of grids in the domain across all processes. The second global communication call is used by each process to determine their access region in the shared output file. These individual offsets can be determined by the `MPI_Scan` method. This method computes the partial reduction of the former global sum. In other the words, the method returns the partial sum of the value of all preceding processes.

The dataset containing the grid data is a two dimensional array. Each row contains the data of a single grid. The global sum allows each process to generate this data set collectively, the scan provides the individual offsets into the array. For example, process zero always has offset 0. It writes its grid data into the first n rows of the collective array, with n being the amount of grids it holds. Process two has offset n, it independently writes its grid data into the rows of the array starting from $n + 1$. Since the root grid is by design always held by process zero, it is always written into the first row of the data array. This makes it a convenient starting point for traversing the hierarchical domain tree for the selective visualisation method mentioned above.

In conclusion the I/O functionality of the framework supports checkpointing and two ways of visualising the grid data using the HDF5 API. The premise of the framework, to reduce global communication as much as possible is still adhered to, using independent file access to a shared file. However, there are two global communication calls necessary. One for the initial generation of the file and a second to determine the size of the grid data and each processes' individual access region for each write.

3.8 Summary

When first conceiving the present work, storing and updating a central repository of the domain structure to carry out a global partitioning technique was identified as most pressing bottleneck. Therefore, the initial idea was to introduce a repartitioning scheme based on a local method to alleviate the bottleneck. However, during the course of the work it became clear that any kind of globally shared data structures are detrimental to performance when the domain structure grows or more processes are involved. With the main performance gain in computing architectures in the last few years stemming from an increase in parallelisation, other parts of the numerical pipeline will similarly exhibit scalability issues which need to be addressed. Therefore, even though the main focus of this thesis lays on the dynamic repartitioning as the most crucial bottleneck, all other parts of the simulation pipeline, with globally shared structures were introduced and, wherever possible, local alternatives have been proposed and implemented.

The basis of this effort is the decentral data structure, which is exclusively distributed. On top of this structure a neighbourhood model has been proposed. This neighbourhood model is deliberately reduced to the bare minimum to support the requirements of a numerical simulation. An extended neighbourhood model, which allows the use of higher order stencil operations and also a higher order discretisation of the model for repartitioning could be conceived in the future though.

The main modules of the framework are designed after current methods, however they have been revisited and altered considerably according to the boundaries set by the choice of available metadata and the use of a communication cycle that only includes neighbouring processes given by the neighbourhood model. These modules include the initial domain generation, the AMR, the solution procedures for the differential equations and the I/O facilities. Furthermore, the AMR module is subject to an additional 2:1 balance constraint to limit interpolation errors between different neighbouring grids with different spatial resolutions. Finally, an asynchronous solution method has been explored and implemented as a fitting extension on top of the decentral structure.

Chapter 4

Dynamic Partitioning

After a defined number of iterations towards a solution the AMR pipeline is executed. Processes evaluate whether a local resolution in the domain is still within acceptable boundaries in terms of accuracy and speed. If higher accuracy is needed, a refinement is ordered. Where accuracy is sufficient, a coarsening might be applicable. In the present distributed data structure each node of the tree is discretised using a grid with the same, predefined number of grid points. When a node, respectively the grid is refined, a new set of child grids is generated, increasing the resolution of the original domain. In turn, when grids are deleted, the domain is solely represented by the parents which enclose a greater spatial region with the same amount of grid points per grid, in turn, decreasing the resolution of the region. New grids are generated on the process to which the respective parent is assigned to. Consequently, the number of grids on this process increases. Vice versa, a coarsening lowers the number of grids on a process. In time, this leads to imbalances in grid distribution and decreases the efficiency of the solution procedure. Furthermore, new grids may also increase the surface of a partition and lead to a higher number of neighbouring partitions, in turn, leading to an increase in neighbour processes that add communication overhead.

To alleviate occurring load imbalances and to minimise neighbourhood connections, a repartitioning becomes necessary. In other words, the distribution of grids to processes has to be evaluated anew. As discussed in chapter 2, there is a variety of global and local methods for partitioning the domain. In general, global methods provide overall better partitions, their main drawback however, is the need for up-to-date structure information of the complete domain decomposition. More accurately, geometry-based methods need the location and weights of the decomposed primitives, graph-based methods require the graph with the primitives as nodes and the edges representing their dependencies among each other.

The number of partitions is equal to the number of participating processes, the number of primitives is generally much higher (see also section 2.2). Having a large number of primitives allows easy distribution and balanced workloads among all partitions, while at the same time the bookkeeping effort of storing and managing the primitives, their weight and their location or their dependencies, is high. With the continued increase in computing power, mainly based on increased parallelism through more processors, the number of primitives a global method has to manage will become no longer feasible.

The only techniques to manage such a number of primitives without memory or synchronisation bottlenecks are local methods. However, they themselves have a number of disadvantages. As workload can only be transferred locally, the number of iterations required to reach a balanced state might be high. Furthermore, guarantees about the quality of the partitioning are hardly possible. And lastly, partitions might fragment, leading to an increase in communication costs. Nevertheless, local methods are currently the only approach which seems practical to be employed for dynamic partitioning applications, which should be deployed to current and future supercomputers. As such, the framework employs a dynamic repartitioning module based on a local diffusion model to balance workloads after the domain has been altered through AMR. The characteristics and benefits of the diffusion model are explained in the following.

4.1 Diffusion Model

Berger et. al. state that "the computational costs inherit properties of the physical system" when applying AMR [25, 24]. More specifically, the resolution of a region is adapted to the physical system one tries to solve. In turn, the number of grids to reach the desired resolution also follows the properties of the physical system. It seems like a sensible choice, to also use a model based on the physical system to distribute these grids among the participating processes. Therefore, a diffusion model was chosen for the dynamic repartitioning, hoping it would lend itself well to diffusion problems or such with diffusive terms.

Local diffusion methods have seen some application within multilevel methods to improve partitioning locally. However, until recently, they have not been applied exclusively for large-scale applications. Recalling the goals of the repartitioning, a diffusion approach looks like a sensible choice. It can be shown, that diffusion processes tend towards a balanced state. Satisfying the first goal of balanced and adjusted workloads. Furthermore, the convergence of this approach is accelerated by making use of the data structure, which allows to transfer workload not only in the spatial, but also through the hierarchical dimension. The diffusion model to balance loads is discussed in section 4.1.2 in detail.

Minimal communication between partitions is the second goal of a good partitioning. One tries to minimise the surface area of a partitioning and ensure it stays continuous spatially as well as through the hierarchy. The later meaning all grids within a partition are connected either directly or indirectly through other grids in the same partition. The diffusion model operates on a model of the communication structure, as workload is only transferred between processes holding neighbouring grids. However, the diffusion itself only balances workloads without optimising neighbourhood connections or keep partitions continuous. Therefore, the second part of the repartitioning module is the determination which primitives are actually transferred to balance load while at the same time keeping a favourable neighbourhood structure. There are a few possible approaches to determine which grids should be transferred to which neighbour process. Additionally, there are cases when transferring a grid leads to an increase in neighbouring processes. Possible different selection approaches as well as weighing opposing goals are discussed in detail in section 4.2.

Nevertheless, the diffusion rebalancing won't be able to compete with global methods in terms of overall workload balance and minimal communication, especially when a low order diffusion or limited load diffusion steps are applied. An SFC distribution for example, simply cuts a linearised list of primitives into equal sized chunks to form partitions. There cannot be a better balance. Furthermore, the Z-order curve used for the initial partitioning has been shown to give a neighbourhood preserving distribution with contiguous partitions and minimal surface areas. Starting from this partitioning, the diffusion model tries to keep the favourable properties of the initial partitioning.

However, when it comes to a fast and scalable method with minimal redistribution, the diffusion approach has clear advantages. The amount of neighbours a grid is connected to is bounded. Consequently, a process with a maximum amount of grids has a bounded number of neighbour processes with which loads can be exchanged. This is completely independent from the total number of grids or participating processes, which promises a very good scaling behaviour of the method. Finally, as the redistribution follows a diffusive model, only a limited number of grids are transferred from one neighbour process to the next. In comparison to global methods, where large quantities are shifted within all participating processes, the redistribution costs of the local method will be much lower.

The remainder of the chapter is structured as follows. First, the workload model is illustrated and possible improvements are discussed. Next, the diffusion model is introduced. Afterwards, the different possibilities of determining which grids are migrated to which neighbour process are discussed. To conclude the repartitioning module, the actual migration of grids is detailed. Since the receiving process might not actively communicate with neighbours of the migrating grid, updating the neighbourhood relations is not trivial in the distributed data structure. This is addressed here as well.

4.1.1 Workload and Communication Model

In the distributed data structure, each node of the tree is discretised using a grid with an equal amount of degrees of freedom, regardless of the physical extent the grid covers. This means, given homogeneous processing elements, a single sweep of an iterative solution procedure over all degrees of freedom should take approximately the same amount of time.

In the basic Jacobi solution procedure, only the leaf grids should contribute towards the load measure on a process, as only these grids compute sweeps over their grids. When adaptive smoothing is applied within the multigrid solution procedure, the picture becomes more involved. Based on the depth of the respective grid, the amount of adaptive smoothing steps in the current implementation is doubled for every depth less. This means that smoothing on coarser grids is more expensive than finer ones by a factor depending on their depth. An individual exception marks the root grid on depth $d = 0$. The SOR method is executed until the approximation of the solution is sufficiently small or a defined maximum number of sweeps has been executed. Depending on how many iterations are needed, the cost of relaxation on the root grid is variable, although with an upper bound. To estimate the cost, an error estimator could be employed. Given the fact that the root grid is only a single grid, the impact of its cost on the overall simulation time is deemed miniscule and an exclusive treatment seems unwarranted.

In its current implementation, each grid, regardless of depth is said to contribute one unit of computational work w towards the load per process. A first improvement, could be to include the number of adaptive smoothing steps into the work contribution. However, to get a first overview over the advantages and drawbacks, a simple model of the workload suffices. The famous quote by Sir Tony Hoare, "premature optimization is the root of all evil" also supports this viewpoint.

The communication model can be similarly involved. It is mainly needed for determining the best migration candidates to minimise the communication frequency with grids assigned to neighbouring processes. Again, the simplest model weights a spatial edge (the relation between two grids on the same level) similar to a hierarchical edge (the relation between parents and children). However, this model can be extended manifold. A pure Jacobi procedure on a uniform refinement only entails spatial ghost layer exchanges on the leaf grids. Therefore, in this case, spatial edges of coarser grids and hierarchical edges should not be considered at all when deciding which grids should be transferred. In adaptive refinement structures, more hierarchical and spatial edges affect the communication volume. These are the edges involved in forwarding ghost layer data when leaf grids miss spatial neighbours on their refinement depth. Finally, in the multigrid procedure all spatial and hierarchical edges impact the overall communication volume, because of the residual and error transfer between depths and the smoothing on coarser levels which entails horizontal communication.

Not all spatial edges can be weighted equal. Depending on the depth of a grid, a coarser grid will adaptively smooth more often. After every smoothing a horizontal ghost layer exchange happens, which means the spatial edges of a coarser grid have a larger impact on communication volume compared to a finer grid. Similarly, not all horizontal edges can be weighted equal, too. While restriction and prolongation transfer the same amount of data, a ghost layer forwarding transfers a different amount of data. In both cases the first operation transfers an amount of data related to the amount of local cells of the parent grid. Ghost layer forwarding transfers only ghost layer data, which naturally has one dimension less. Furthermore, not all ghost layers of a grid have equal size as the resolution in each dimension is variable. Finally, the network configuration also plays a role in determining the weight of an edge. An edge connecting two grids, assigned to two processes running on two processing elements on the same computing node, is much cheaper than if the two processors are on separate nodes, connected trough ethernet. Depending on the network configuration of the computing cluster as well as the current usage, connection speeds might be highly variable, which in turn affects the weight of the edges. In conclusion, the weight of spatial and hierarchical edges are neither consistent between themselves nor between each other. However, modeling these intricate weights seems excessive at this stage of the framework. Each edge's weight is assumed to be $e_w = 1$. There is however the possibility to prefer one edge type over the other. This will be explained further in section 4.2.

The problem of establishing workload and communication models becomes even more involved when comparing communication volume given by the amount and weight of inter-process edges and the workload given by the amount of grids per process. In other words, weighting both models against each other, a better balance against possible higher communication costs, requires not just a relative comparison within the models, but absolute costs. Again, at this point, there is no definitive answer as both models are not comparable currently. Section 4.2 however, gives a few possible strategies that are also implemented in the framework.

A more accurate combined workload and communication metric could be concluded from either the non-idle runtime of each process during the solution process, or from the amount of asynchronous iterations of each process of an asynchronous iterative method. These metrics give an exact value for the current load each process handles, including all influence factors described above. Nonetheless, the insight into which factors contribute which load cannot be gathered directly. Though, as a starting point to get a broader picture and evaluate different balancing and distribution strategy, these metrics could prove viable.

4.1.2 Load Balancing

The diffusion model for dynamic load balancing that is used in the current implementation, was first described by Cybenko in [66]. The basic model is stated as follows:

$$w_i^{(t+1)} = w_i^{(t)} + \sum_n \alpha_{ij}(w_j^{(t)} - w_i^{(t)}) + \eta_i^{(t+1)} - c, \tag{4.1}$$

where the work distribution at time t is quantified as a vector $w^{(t)}$, and $w_i^{(t)}$ is the number of tasks to be done by process i at time t. The size of vector w is the amount of processes in the system. The summation is over all processes in the network n. In the original model, $\eta_i^{(t+1)}$ represents the new work generated at time t for processor i and c is the constant amount of work that can be accomplished by any process between time t and $t+1$. The non-negative constant α_{ij} determines the ratio of workload swapping between process i and j, with $\alpha_{ij} = 0$ if i and j are not connected. Dropping the inhomogeneous terms from 4.1 results in the static model:

$$w_i^{(t+1)} = w_i^{(t)} + \sum_n \alpha_{ij}(w_j^{(t)} - w_i^{(t)}) \tag{4.2}$$

and the amount of load exchanged between two processors with:

$$\alpha_{ij}(w_j^{(t)} - w_i^{(t)}) \tag{4.3}$$

Cybenko determines α from the completely known structure of different static networks. For the framework's dynamic structure, the extension of the model from Boillat, proposed in [27, 28], is used. Only local knowledge of the connections of each processes is required here to compute α. Consider $G = (N, E)$ as a non directed, connected, acyclic finite graph. The vertices of the graph are labelled with numbers $(N = 1, 2, ..., n)$. The number of edges a vertex has to another vertex, also called the degree of a vertex i, is denoted by v_i. The canonical valuation of the Graph G is given by:

$$c_{ij} = \frac{1}{max(v_i, v_j) + 1} \tag{4.4}$$

Boillat has shown that using the canonical valuation for α in 4.3, continuously exchanging loads converges strictly asymptotically towards a fixed point, i.e. a balanced state. For inhomogeneous systems, an additional way to compute the constant can be found in [168].

As already established, the workload w_i is set as the number of grids held by process i. The degree of process i is given by the number of neighbours of said process. One communication cycle of the process data exchange method described in section 3.3.1.4 is used to exchange the current load and the current degree between all neighbouring processes. This allows the use of the canonical valuation 4.4 as a dynamic α, and consequently, the computation of the amount of load one process needs to transfer to its neighbours to balance.

An issue with this method is that the load exchange cannot account for load fractions. As only whole grids can be migrated from one process to the next, loads can only be transferred in increments of one. Currently, the load to be migrated is rounded to the next integer, which introduces a balancing error. Therefore, processes tend to either transfer too much load when rounding up, or too little when rounding down. This effect is especially detrimental for processes with a high number of neighbours, i.e. a high degree and consequential a very low α. Since the difference in weight between neighbouring processes is multiplied by α, these high degree processes tend to accumulate high numbers of grids without transferring them to their underloaded neighbours. The rounding down prevents them to balance fractional loads. However, as soon as the loading becomes to high and the load fractions cross the threshold of 0.5, suddenly, a lot of grids get migrated. This transfer overcompensates in the opposite direction, because due to the rounding up, the number of migrated grids is higher than the total computed load transfer.

One possible remedy for the overcompensation could be to never allow more grids to be transferred as the total load transfer, while preferring processes with a higher target fraction as receivers. However, this does not solve the rounding issues conclusively. Only if the granularity of the decomposition is fine enough, such that rounding hardly matters could this problem be solved. A very fine decomposition is however detrimental to performance on the other hand. Consequently, more research is needed to derive a good determination of the granularity of the decomposition, weighing the load balance against the solution procedure's efficiency.

4.2 Target Selection

Up to now the diffusion model only considers the workload balance, i.e. a balanced number of grids per process, while disregarding the minimisation of communication links to other processes. The goal is to have as few neighbourhood connections of grids across process borders as possible and, if possible, cluster the inevitable connections to a limited number of processes. This is also meant when talking about the minimisation of the surface area of a partition. In the same regard, it is important to keep partitions contiguous in spatial and hierarchical direction. Additionally, if a weighting of edges is available, prefer heavy edges with a high communication volume for intra-process connections and light edges, entailing a low communication volume for inter-process communication. As the communication cycle only considers neighbouring processes, possible targets for a migration are said neighbours. This implicitly helps in keeping favourable neighbourhood relations.

There are two questions to be answered here. The first is which grids should be migrated to which neighbour processes to keep the inter-process communication volume and cost as low as possible. The second question is, if a migration should be forced at all costs, even if the migration leads to a worse structure, for example a fracturing of the partition. These questions cannot be answered conclusively as they are highly problem dependent. Therefore, the current implementation gives a variety of choices based on the current communication model.

All target selection methods are subject to two constraints. First, the root grid can never be moved from the process with MPI-rank zero. The root grid serves as starting point to traverse the complete tree top-down. Several methods depend on this defined starting point. Second, a process is never allowed to migrate all grids to their neighbours. A process with no grids has no neighbour processes any more, is excluded from the communication cycle and can therefore never gain any grids any more. It would be idle for the remainder of the runtime. In the formulation of the load balancing this case theoretically cannot happen, nevertheless, the rounding issues when computing a load fraction could cause all grids to be migrated in an overcompensation.

Currently, there are seven different target determination methods available. For completeness, there is a method that distributes grids randomly, which is not used in practice. All other methods are based on the degree of a grid with its current process and with a possible target process. The degree in this context is nothing else than the amount of neighbourhood relations with other grids assigned to the present or to the remote process. The first target determination method weighs the overall degree of each grid, that means the combined amount of spatial and hierarchical connections on the target process. The second and third methods prefer the hierarchical degree over the spatial degree and vice versa. The local degree is not considered in these methods. If after several migrations, a process should receive more grids, but there are none available with a remote degree greater than zero, meaning they have no relations to any of the remote grids, no grids are transferred to this neighbour process any more. This would lead to a non-contiguous partition and is therefore relinquished to the detriment of a better balance.

The other three methods do exactly the same as the former three with the difference that they transfer grids even if the remote degree is zero. Meaning they accept a worse communication structure in favour of a better balance. The grids to transfer in this case are chosen in opposite order of their local degree. That means, the grid with the lowest local degree is transferred first, then the second lowest etc.

Herafter, a summary of the six main target determination is given:

HD - highest degree target determination
Grids are sorted according to their overall highest remote degree and send from highest to lowest degree if the respective target process still needs to satisfy the calculated load transfer. Grids are never transferred to processes where they have a remote degree of zero, even if the respective process still requires grids to equalise load.

HDF - highest degree target determination with forced migration
Same as HD, except grids are also transferred to processes where they have a remote degree of zero. After all grids with non-zero remote degrees have been transferred and the load balance is still not satisfied, left over grids with remote degree zero are taken into account. The selection is based on the local degree of the grid, starting with the grid with the lowest local degree.

HHD - highest hierarchical degree target determination
Grids are sorted according to their hierarchical remote degree first and according to their spatial remote degree second. The sorted list is then traversed and grids are send to their respective target processes, if they still need grids according to the calculated load transfer. Grids with a remote degree of zero are not transferred to satisfy the load balance.

HHDF - highest hierarchical degree target determination with forced migration
Same as HHD, except grids are also transferred to processes where they have a remote degree of zero. Again, the local degree in reverse order is used to determine suitable grids to send to satisfy the load transfer given by the diffusion model.

HSD - highest spatial degree target determination
Counterpart to HHD. Grids are sorted according to their spatial remote degree first and according to hierarchical remote degree second. Migration is based on traversing the sorted list. Grids with zero remote degree are not transferred to satisfy load balance.

HSDF - highest spatial degree target determination with forced migration
Counterpart to HSDF. Initial migration is determined by sorting all grids according to HSD. To satisfy the remaining load transfer, grids with zero remote degree are taken into account and are send according to their local degree in reverse order.

In Algorithm 9 the technical implementation is illustrated. The first step is to iterate through all neighbour processes and calculate the degree of each grid towards this neighbour process. This is done by checking the remote neighbour references, i.e. the UIDs which conveniently also encode the neighbour MPI-rank. A tupel containing a reference to the grid and the respective neighbour rank is added to a candidate array if the degree is larger than zero. In a second step, the candidate array is sorted given the respective predicate. These are, as mentioned above, either the highest overall degree, or a preference of the spatial or hierarchical degree. The sorted array is now traversed, yielding the current most favourable grid to transfer. If the respective neighbour still needs grids to balance its load, the grid is migrated, a migration counter is increased, the load requirement of the neighbour process is decreased and the grid is removed from the candidate array. A grid may be present multiple times in the candidate array, once for each neighbour process with positive neighbourhood relations. Therefore it is important to remove all references

Algorithm 9: Target determination method

```
// Generate an array of transfer candidates
foreach neighbourProcess do
    foreach grid do
        neighbourDegree ← getNeighbourDegree (grid, neighbourProcess);
        if neighbourDegree > 0 then
            candidateArray.append (grid, neighbourProcess);
    end
end
// Sort the array according to the preferred predicate
sort (candidateArray, sortPredicate);
// Iterate through the candidate list and migrate grids
foreach candidate in candidateArray do
    if candidate.neighbourProcess.loadTransfer > 0 then
        migrate (candidate.grid);
        migrationCounter ++ ;
        candidate.neighbourProcess.loadTransfer −− ;
        candidateArray.remove (candidate.grid);
        if migrationCounter = totalTransfer then
            return;
end
// Optional for forced migration
// Generate an additional candidate list of left over grids
foreach grid do
    candidateArray.append (grid);
end
// Sort the array according to the local degree
sort (candidateArray, smallerLocalDegree);
// Migrate left over grids until the load transfer is satisfied
foreach neighbourProcess do
    if neighbourProcess.loadTransfer > 0 then
        migrate (candidateArray.grid);
        candidateArray.grid ++;
        candidate.neighbourProcess.loadTransfer −− ;
end
```

of the specific grid. The algorithm concludes if either the candidate array is empty or the amount of completed migrations has reached the total number of grids that should be transferred to reach a balanced state.

The second part of the algorithm is only executed in the target determination methods with forced migration, meaning those which also transfer grids with remote degree zero to satisfy the load balance. The candidate array is filled again with the left over grids. Similar as before, the candidates are sorted. This time according to the local degree, because all remote degrees are zero. A traverse through the array now gives the grid with the current lowest local degree. Each neighbour is checked whether the required number of grids have been migrated already, if not, the current grid is migrated to this process. As no duplicates are in the candidate array at this point, a removal is not necessary, but the iterator pointing to the current element must be increased. After the migration, the load requirement of the neighbour process is decreased. The method concludes when all neighbour processes have gotten the required number of grids to balance or the candidate list only has a single entry left (to prohibit no grids being left on the process).

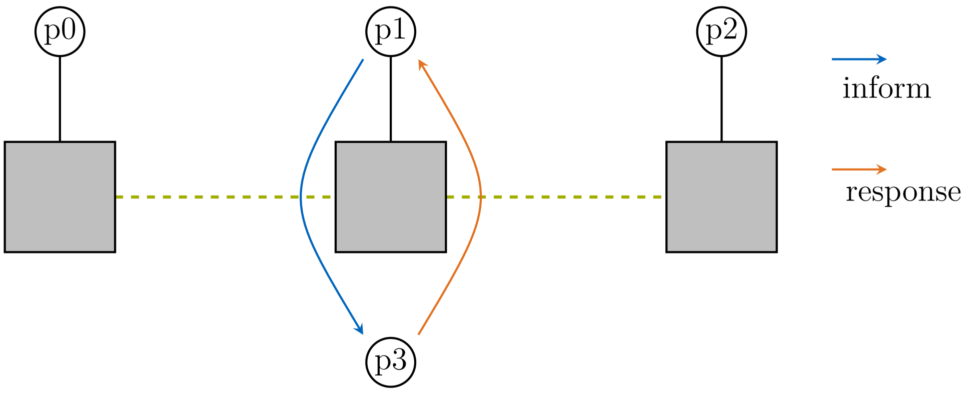
This concludes the target determination methods implemented in the framework so far. One additional option using the current communication model would be to weigh local and remote degrees against each other. However, a broader understanding is necessary to estimate the repercussions of weighing a better communication structure against a more favourable load balance and to make an informed decision when one should take precedence over the other.

4.3 Migration

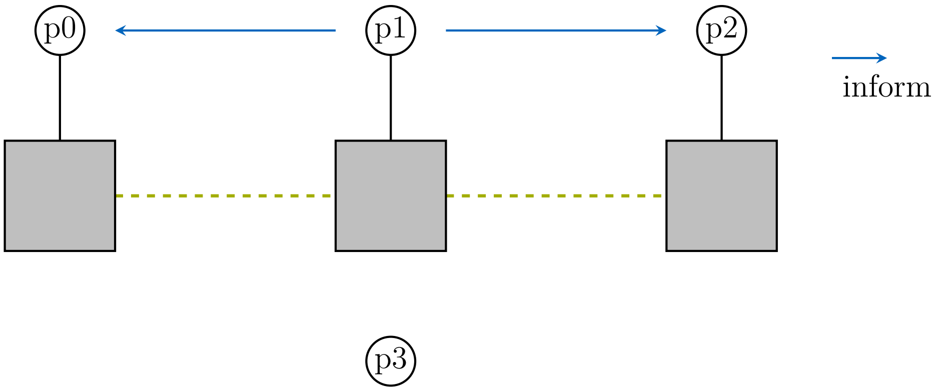
To conclude the dynamic repartitioning module, the migration algorithms are introduced. After the diffusion load-balancing has determined the amount of load to be transferred, and the chosen target determination method has selected the target for the respective grids, the grids need to be actually transferred. This includes the grids metadata and its simulation data. In addition, all neighbourhood references held by other grids must be updated accordingly. Due to the decentral grid distribution, the metadata updates must be issued by the original holder of the grids. Before the final migration of a grid, processes do not necessarily communicate with neighbours of the grid to be migrated and cannot issue update queries to the respective processes.

The complete migration and update method is a three step process. An example is illustrated in Figure 4.1. The setup consists of four process, p0 to p4, whereas p0, p1 and p2 each hold a grid with neighbourhood relations among them (dashed green lines). p3 does not necessarily have any neighbourhood connection to p0 and p2, though it has neighbourhood relations with p1 (not pictured). The goal is to migrate p1's grid to p3.

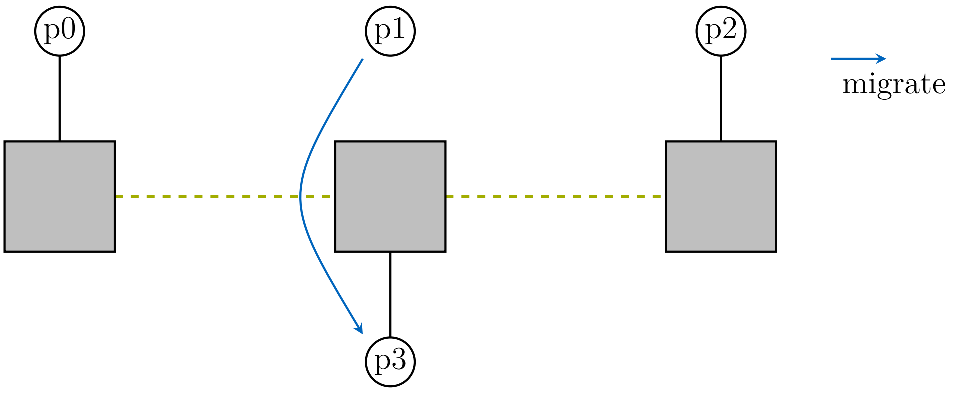
The new grid identifier (GID), as part of the UID of a grid (see section 3.1.4), can only be determined by the target process. Therefore, in a first step, the original process informs the target process how many grids it is going to receive. The target process then virtually generates new GIDs and sends them back to the origin process. With the new GID, the



(a) Origin process informs target process how many grids are about to be transferred. Target process sends back new GIDs.



(b) Origin process informs all neighbours of metadata update.



(c) Grid is migrated.

Figure 4.1: Three stage migration and metadata update method.

new MPI-rank and the static hash of the grids, new UIDs can be constructed on the origin process. This back and forth communication is illustrated in Figure 4.1a. In the second step, the new UID is communicated to all processes with grids that hold a neighbourhood reference. The specific grids that need to update their reference are easily determined from the references of the grid about to be migrated. This update is illustrated in Figure 4.1b. In the third stage, the actual migration takes place. Each grid is transferred from origin to target with its metadata and simulation data, pictured in Figure 4.1c.

The migration of grids may have caused a change in neighbourhood structure of the assigned processes. Therefore, after the migration, each process must update their communication links, which is simply achieved by iterating through every neighbour of the held grids and storing every unique neighbour rank. In the figure, this means that now p0 and p3, as well as p2 and p3 are neighbouring processes and must establish communication links. p1 must no longer communicate with p0 and p2.

4.4 Summary

This chapter has introduced the dynamic repartitioning method. The motivation for introducing a repartitioning scheme based on a local method were to alleviate bottlenecks of storing and updating a central repository of the domain structure, which is inherently not scalable. The choice of a diffusion-based approach was made to mimic the physical system in the hopes it would complement the AMR module, which also inherits the properties of the physical system.

The diffusion aims to balance the workload across all participating processes. Here, a single grid contributes one unit of computational load. Given enough iterations, it can be proven that the diffusion-based approach converges. In practice however, only whole grids can be migrated, which leads to some difficulties in load balancing. Some pointers have been given for improvement, however this problem is not yet conclusively solved.

The second major part of the repartitioning module is the target determination. After a measure of the necessary load transfer has been calculated, the question remains as to which grids should be transferred to which target processes. The degree of a grid with respect to a possible target process, i.e. the number of neighbourhood connections with grids assigned to this process was proposed as the main deciding factor to which process a grid should be transferred to. Different variants of target determination methods have been implemented and will be evaluated in the next chapter. Possibilities include weighing overall degree or preferring either hierarchical or spatial connections. Furthermore, these variants can also be extended by allowing or prohibiting migrations to target processes with unfavourable neighbourhood relations, weighing a better load balance against worse communication patterns.

After the load transfer and the targets have been determined, the last part of the repartitioning module is the migration method. It is again limited by the constraints given by the decentral data structure. The necessary metadata update methods have been discussed as well. This concludes the theoretical framework as well as the implementation details of the dynamic partitioning module. In the next chapter, the module is evaluated using a number of different test cases and the results are discussed.

Chapter 5

Evaluation of the Dynamic Repartitioning

In order to show the viability of the implemented dynamic repartition approach, three example test cases have been selected with increasing closeness to real applications. Firstly, a pure artificial test scenario with a moving front, around which refinement takes place is analysed. The second example is modelled after a simplified formulation of a laser powder bed fusion benchmark. A prescribed heat source representing the laser is applied to a metal surface, whereas the heat is dissipated through the material. The strength of the heat-dissipation acts as an indicator for regions where refinement or coarsening is required. Finally, the well-studied benchmark scenario of vortex shedding behind a circular obstacle within a flow is analysed. Here, the incompressible Navier-Stokes equations are solved and the vorticity is used as an indicator to identify regions where the resolution has to be increased or may be decreased.

As a baseline, a pure SFC repartitioning with a Z-Order curve is used. When it comes to workload balance, the SFC returns an optimal result. In terms of connectivity, i.e. the number of inter-process edges and minimisation of the surface area of partitions, the Z-order curve has been shown to produce very good results [10, 265]. However, it has been established that this decomposition cannot be computed regularly during runtime if the system size becomes to large. Therefore, the question is how good the local approach can be comparable to the “optimal” distribution for balance and connectivity.

All test examples were run on Leibniz Supercomputing Centre’s (LRZ) Linux Cluster System [201] on the CoolMUC-2 cluster segment [200]. The specifications of the system are: 812 28-way Intel Xeon E5-2690 v3 Haswell-EP nodes with Infiniband FDR14 interconnect and two hardware threads per physical core. The theoretical peak performance of the segment is 1,400 TFlop/s. The compiler used was the Intel Compiler in version 19.0.

5.1 Test Example - Growing AMR Sphere

The first example is a pure artificial test case for the repartitioning. There is no physical system to be solved. Instead, a hollow sphere is placed inside a cubic domain at its center. With every timestep the sphere is enlarged until the complete sphere encloses the domain. Wherever the sphere surface intersects a cell, the corresponding grid is refined to a defined depth. Vice versa, when the intersection no longer applies, the domain is coarsened and grids are deleted again.

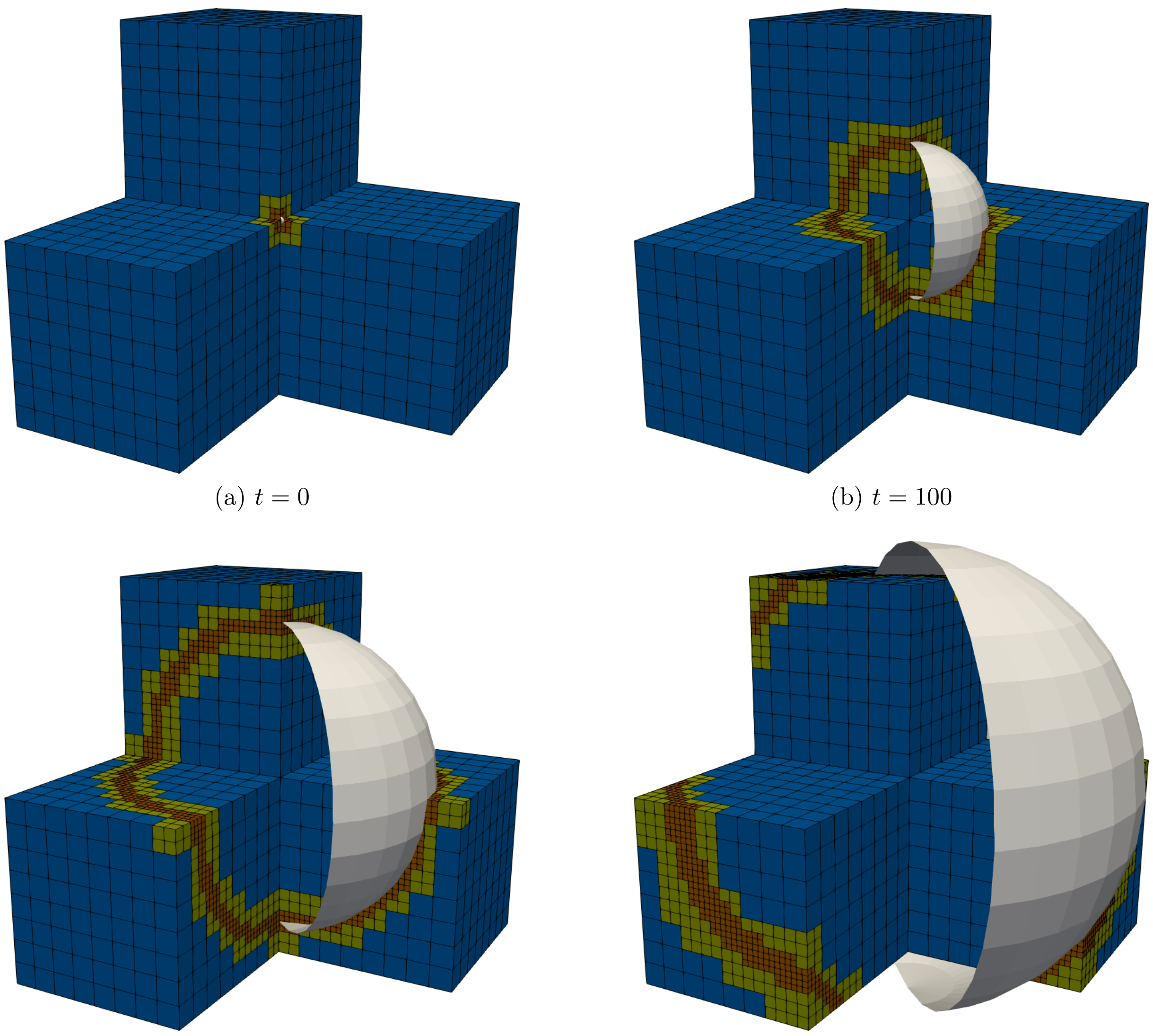
5.1.1 Test Setup

The setup consists of a cubic domain of unitless size $1 \times 1 \times 1$. The domain is decomposed using a subdivision of two in every cardinal direction, generating the classical octree structure with eight children per refined parent. The domain is uniformly refined to depth $d = 4$ resulting in 4,608 total grids and 4,096 leaf grids on the deepest refinement level. Now the sphere is introduced into the domain at the center $(0.5, 0.5, 0.5)$ with unitless radius $r_{init} = 0.01$ and an adaptive refinement is performed to depth $d = 6$, where the sphere surface intersects with the cells of a grid. This is the case for the eight grids directly surrounding the center on depth $d = 4$ resulting in 64 children on depth $d = 5$ and again on the eight grids surrounding the center on depth $d = 5$, resulting again in 64 more children on depth $d = 6$. Consequently, the initial refinement results in 4,736 total grids and 4,208 leaf grids in the domain.

Over the course of the runtime, the sphere grows with a normal velocity of $vel_{normal} = 0.002$ per pseudo timestep. This means, the radius of the sphere grows by 0.002 every step. After exactly 429 timesteps, the radius of the sphere outgrows the cubic domain. At $t = 429$ the radius of the sphere is 0.868 The locations with the largest distance from the center of the domain are its corners with a distance of $\frac{\sqrt{3}}{2} \approx 0.866$. Wherever, the sphere intersects a grid, it is refined down to depth $d = 6$. After the sphere no longer intersects a grid, the AMR module tries to coarsen the domain back to depth $d = 4$. However, the 2:1 balance constraint must be adhered to, which results in a gradual coarsening around the sphere's surface to depth $d = 5$.

Figure 5.1 shows a cutout of the domain configuration overlaid with an STL representation of the sphere at timesteps $t = 0$, $t = 100$, $t = 200$, and $t = 300$. This illustrates the gradual growth of the sphere and the adaptive refinement and coarsening around the sphere's surface. The illustration depicts only leaf grids, with grids on depth $d = 4$ in blue, grids on depth $d = 5$ in green and grids on depth $d = 7$ in orange.

Figure 5.2 shows the evolution of the number of grids during the runtime of the test. At $t = 0$ the domain has 4,736 grids in total. As the sphere grows the number of grids gradually increases until $t = 253$, where the number of grids in the computational domain reaches its peak at 56,969. This is shortly after the sphere is at its largest, while still completely within the cubic domain at $t = 249$. Afterwards, the number of grids gradually decreases until the cubic domain is completely enclosed by the sphere and no grids intersect with its surface any more. In the end, the domain is back to its initial state, uniformly refined to depth $d = 4$ and 4,608 grids exist in total. During the course of



(a) $t = 0$ (b) $t = 100$ (c) $t = 200$ (d) $t = 300$

Figure 5.1: Cutout of the computational domain overlaid with the sphere in STL format at timesteps $t = 0, 100, 200, 300$. Only leaf grids are depicted, $d = 4$ in blue, $d = 5$ in green and $d = 6$ in orange.

the test case, the sphere has intersected every grid once. Therefore, the domain has been discretised completely to depth $d = 6$ at one time, resulting in 294,912 unique grids that have existed in total. Apart from the pure artificial nature of the test case, the evolution of the amount of grids in the domain also separates this example from the other two, discussed afterwards.

In the following, the distribution of the SFC partitioning is compared against the diffusion-based partitioning with the various target determination methods. The initial partitioning is computed using the Z-order SFC for all testcases. The AMR module evaluates refinement and coarsenings after every timestep. Afterwards a repartitioning is executed. Each case is run on 32 nodes of the CoolMUC-2 cluster, with 28 cores each. Each core runs a

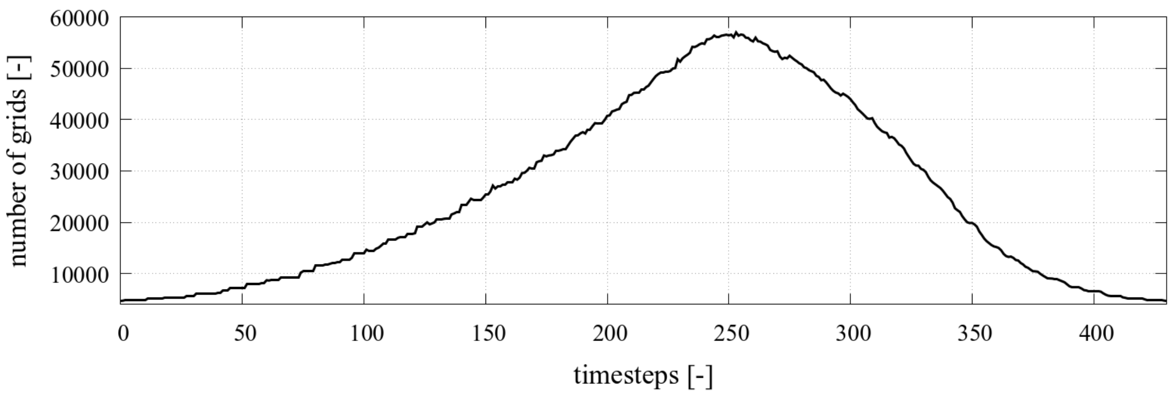


Figure 5.2: Number of grids in the domain per timestep.

single process, amounting to 896 participating processes in total.

5.1.2 Workload Balance

The first key measure observed is the overall workload balance. That is how balanced the distribution of grids onto the participating processes is. Figure 5.3 shows an overview of the workload balance in terms of the standard deviation σ of grids per process of the SFC partitioning, the diffusion partitioning with highest overall degree target determination (HD) and without any repartitioning (NB).

The average number of grids per process (μ) varies between approximately $\mu = 5$ at the beginning and the end of the test run, and approximately $\mu = 64$ at $t = 253$. This average number behaves exactly as the curve shown in Figure 5.2. As mentioned above, the SFC produces optimal balancing results with standard deviations of always less than one. Also to no surprise, without any repartitioning the standard deviation rises considerably when new grids are generated locally and have no way to be migrated to less burdened processes. The highest standard deviation for NB is $\sigma = 69$ at $t = 280$.

The diffusion results have been gathered by running only a single iteration of the method after every AMR adjustment. Here, the standard deviation rises in the first third of the test run to approximately $\sigma = 10$ at $t = 97$, before dropping to $\sigma = 4$ at $t = 158$ and staying between $\sigma = 4$ and $\sigma = 6$. At the highest overall load during the simulation run at $t = 253$ the relative standard deviation is 8.5%.

To get a better picture of the evolution of the workload balance, Figure 5.4 illustrates the frequency distribution of grids to processes at the three distinct timesteps mentioned above. At $t = 97$, the standard deviation is maximal. The distribution is arguably very poor. 55% of processes hold an amount of grids within one standard deviation from the mean. The rest of the processes hold an amount between one and two standard deviations from the mean. Furthermore, 32% of processes hold either five or six grids. This is the amount they started with, suggesting that these processes have not yet actively participated in grid migration. At $t = 158$ the standard deviation is minimal. The distribution is much better than before. 67% of processes lie within one standard deviation

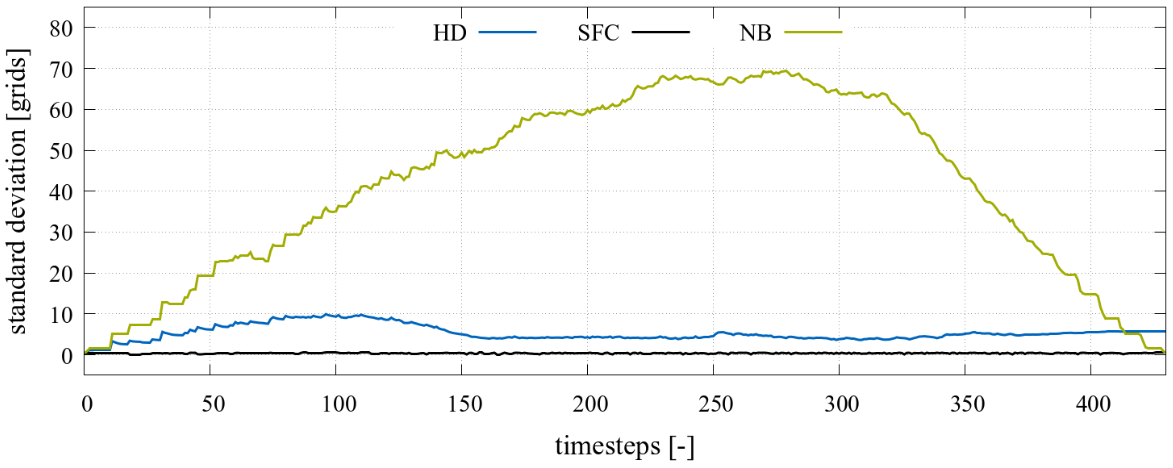
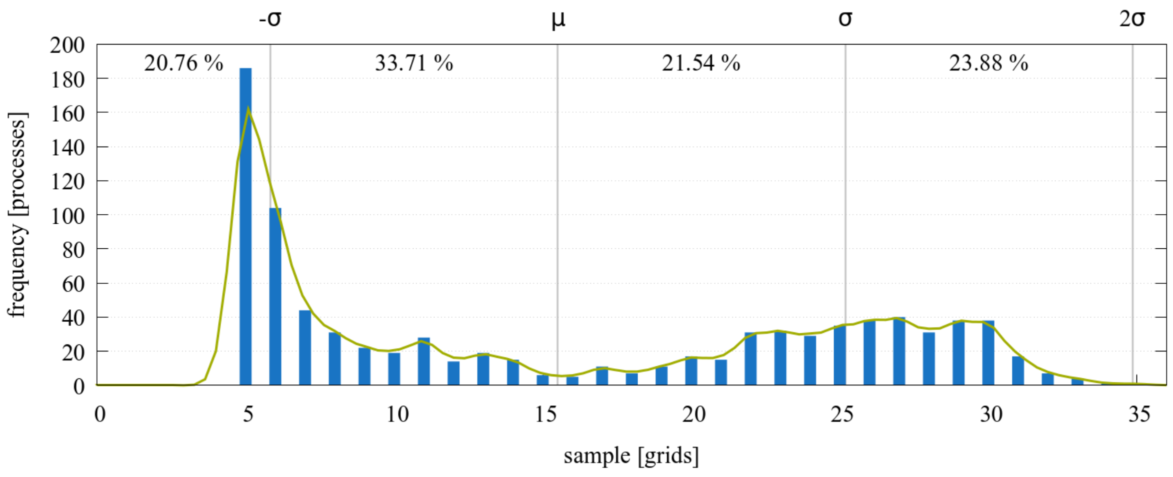


Figure 5.3: Standard deviation of grids per process measured at every timestep with an Z-order partitioning (SFC), without repartitioning (NB) and with a single iteration of the diffusion repartitioning and highest degree target determination (HD).

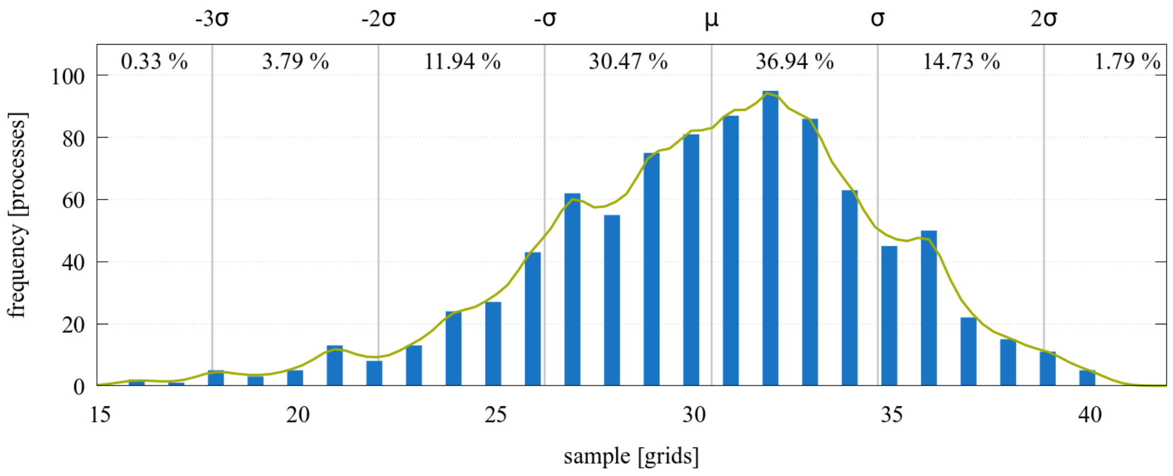
from the mean and 27% lie between one and two. Furthermore, the distribution appears roughly normal distributed, adhering to the three-sigma-rule with two thirds of the processes within one standard deviation from the mean and almost all (94%) within two. The distribution has no outliers as before, suggesting the distribution has reached a good balance. Finally, at $t = 253$ the computational load, that is the total amount of grids in the domain, has reached its peak. The mean number of grids per process is twice as high as before. Nevertheless the standard deviation is still acceptable. The distribution of grids per process is even better, with 72% of all processes within one standard deviation from the mean and 24% between one and two.

After establishing a baseline between the SFC and diffusion repartitioning strategies, a closer examination between the individual target determination methods is performed. First, the highest overall degree (HD) is compared against preferring to distribute grids with hierarchical connections over spatial ones to a target process (HHD) and the opposite, preferring spatial connections over hierarchical ones (HSD). All three methods do not transfer grids to processes without any connections, in other words, they accept a worse workload balance if no suitable grid with neighbourhood connections on the target grid is available. Figure 5.5 shows the standard deviation of the different strategies for every timestep. One gathers that the different methods have a negligible influence on the distribution for the example, since the curves almost perfectly match.

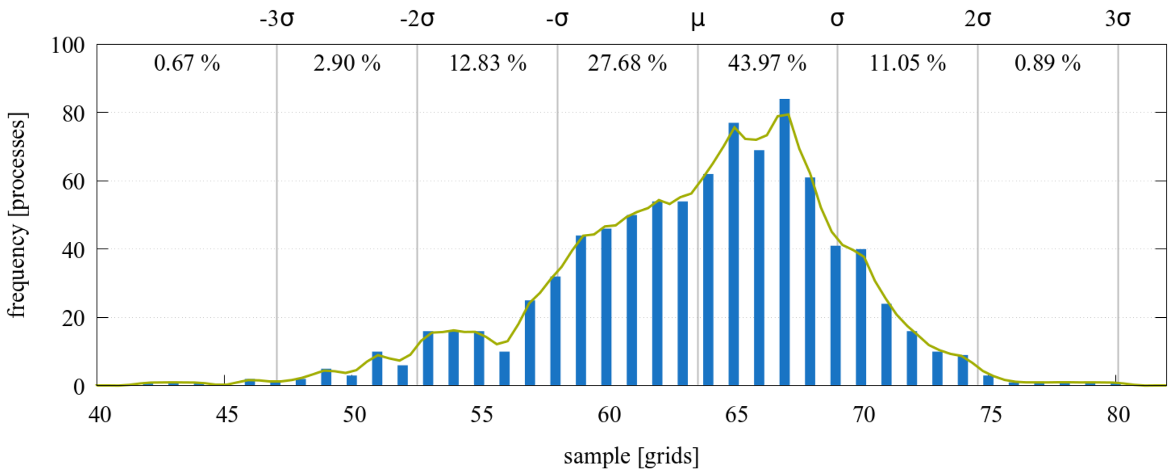
Next, the three target determination methods with forced migration are examined in terms of workload balance. They work similar to their respective non forced counterparts. However, as their name suggests, after all grids with neighbourhood connections have been migrated to a target process, grids without neighbourhood connections are migrated in addition to satisfy the load transfer computed by the diffusion model. In Figure 5.6 the standard highest degree method (HD) is compared to its forced migration version (HDF). For HDF, one observes an initially flatter curve compared to HD that rises to



(a) $t = 97$



(b) $t = 158$



(c) $t = 253$

Figure 5.4: Histogram of the frequency distribution of grids per process at times $t = 97$, $t = 158$ and $t = 253$ with diffusion partitioning and highest overall degree target determination (HD). The histogram is overlaid with the smoothed kernel density estimate using a Gaussian kernel.

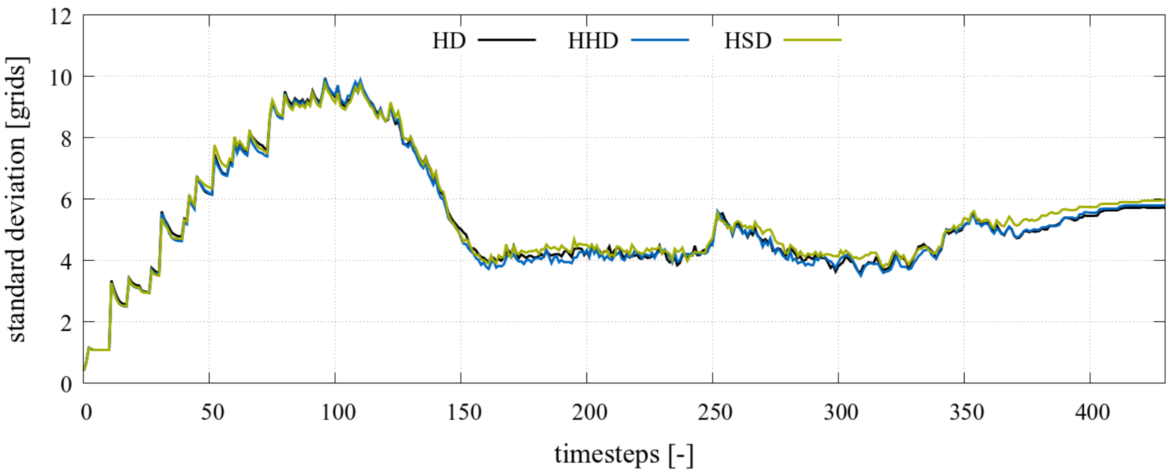


Figure 5.5: Standard deviation of grids per process measured at every timestep with a single iteration of the diffusion repartitioning and different target determination strategies. Highest degree target determination (HD), preferring hierarchical edges over spatial edges (HHD) and preferring spatial edges over hierarchical edges (HSD).

approximately $\sigma = 6$ at $t = 87$. Even though its peak is lower in the first third of the simulation run, it does not recover in the following and the non forced version outperforms the former from $t = 146$ onwards. For the rest of the runtime, the standard deviation remains between $\sigma = 5$ and $\sigma = 7$. At the highest overall load during the simulation run at $t = 253$ the standard deviation is $\sigma = 6.4$, a relative standard deviation of 10%.

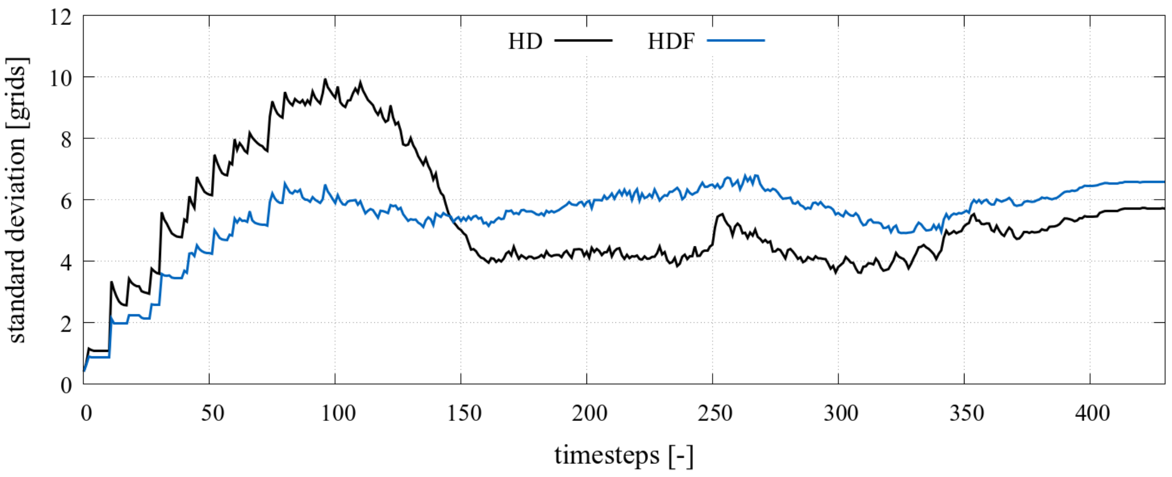
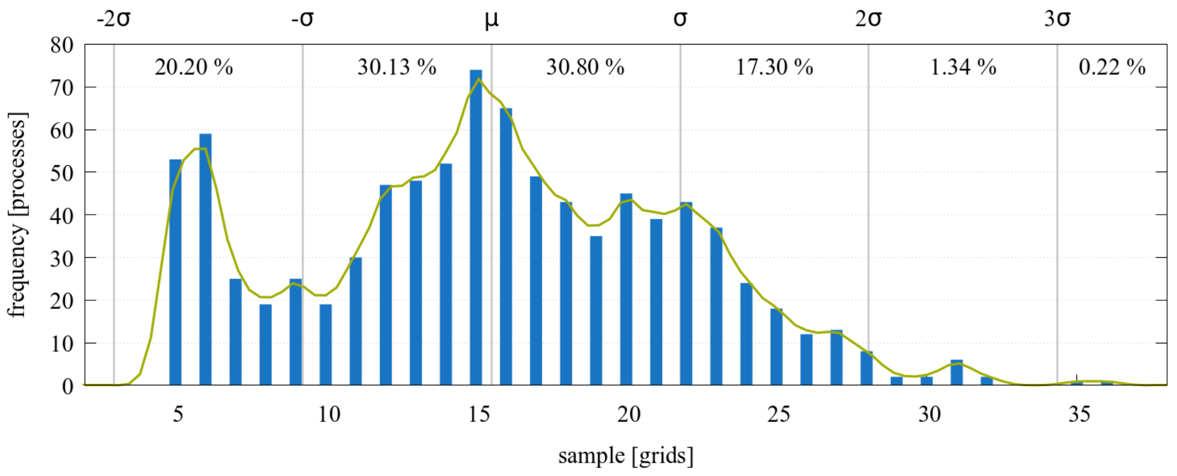
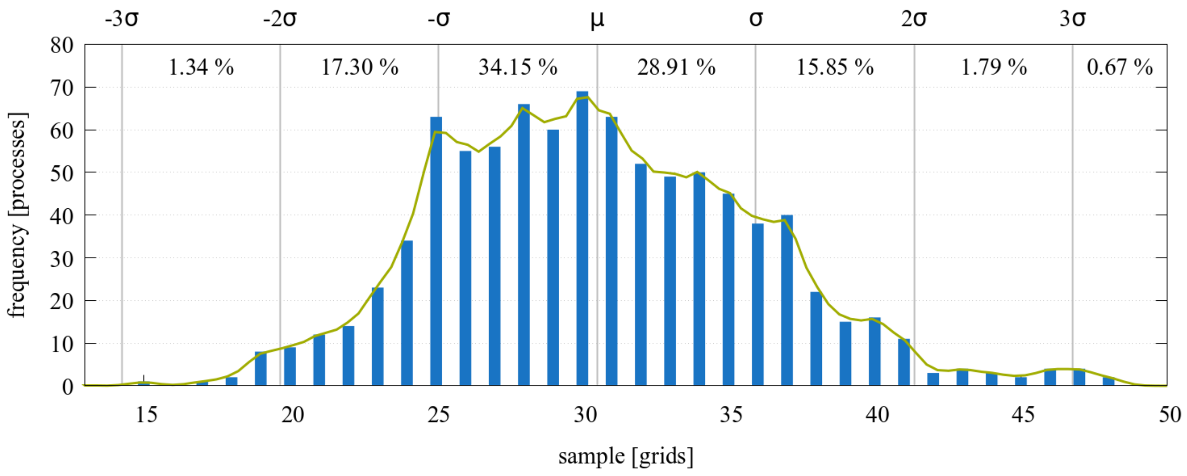


Figure 5.6: Standard deviation of grids per process measured at every timestep with a single iteration of the diffusion repartitioning and different target determination strategies. Highest degree target determination (HD) and highest degree target determination with forced migration (HDF).

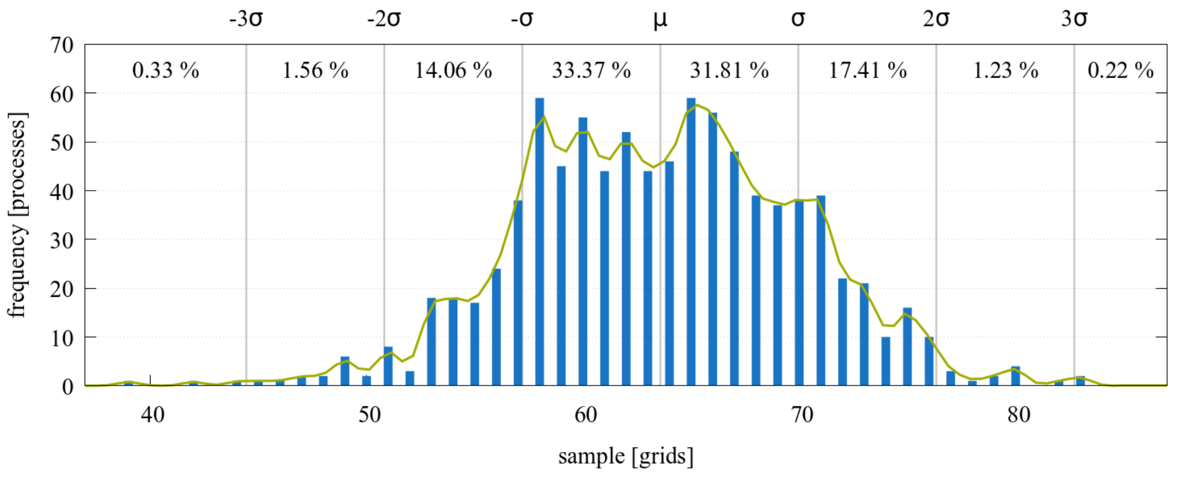
For a better understanding, the evolution of the workload balance is similarly examined for the HDF method at times $t = 97$, $t = 158$ and $t = 253$. This is illustrated in Figure 5.7. At $t = 97$ the distribution is much better compared to HD, with 61% of all



(a) $t = 97$



(b) $t = 158$



(c) $t = 253$

Figure 5.7: Histogram of the frequency distribution of grids per process at times $t = 97$, $t = 158$ and $t = 253$ with diffusion partitioning and highest overall degree target determination with forced migration (HDF). The histogram is overlaid with the smoothed kernel density estimate using a Gaussian kernel.

processes within one standard deviation from the mean number of grids per processes. Furthermore, the amount of processes which have not yet migrated load, i.e. processes with five or six grids is much lower, at only 12.5% compared with the 32% of processes when using the non forced migration version. These results coincide with the expectation that forcing a migration for the benefit of a more optimal workload balance should produce beneficial results. At $t = 158$, HDF produces slightly worse results in overall standard deviation, the distribution itself is comparable however. It is roughly normal distributed with 63% of all processes within one standard deviation from the mean and one third of processes between one and two standard deviations. Compared to HD, there are less processes within one standard deviation (67% for HD), but considerably more between one and two (27% for HD). Furthermore, the distribution is slightly skewed to the left of the mean compared to HD which is slightly skewed to the right. Lastly, at $t = 253$ both HD and HDF have even more similar distributions, with the same percentage of processes within one standard deviation from the mean and between one and two. The main difference is HD is even more skewed towards the right of the mean, whereas HDF is more balanced than before. In conclusion, HDF is able to reach a balance workload faster, due to more possibilities for migration. Both methods tend towards a consistent standard deviation with slightly better values for HD during the remainder of the simulation.

Figure 5.8 illustrates the comparison of standard deviation of grids per process of all target determination methods with forced migration. Similar to the regular versions, no discernible difference can be observed.

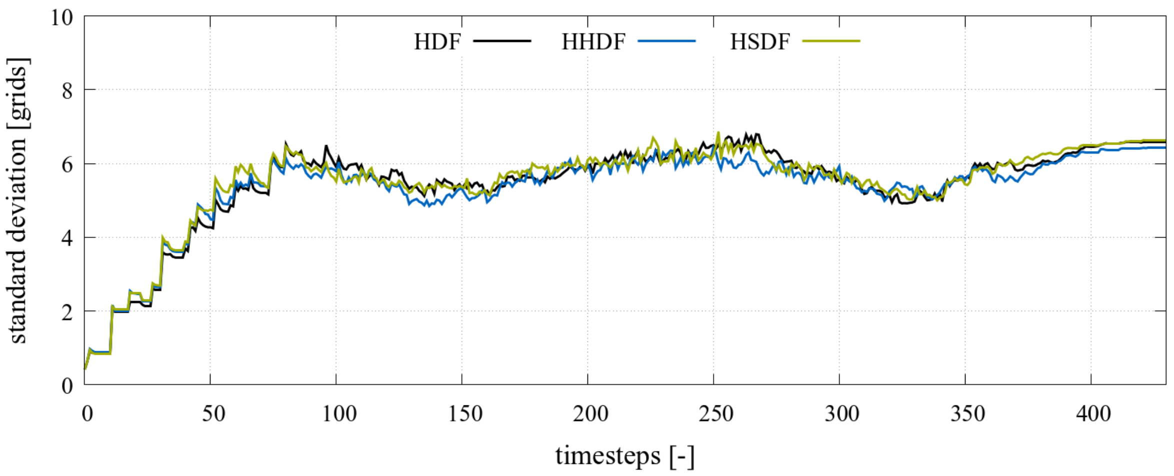


Figure 5.8: Standard deviation of grids per process measured at every timestep with a single iteration of the diffusion repartitioning and different target determination strategies with forced migration. Highest degree target determination (HDF), preferring hierarchical edges over spatial edges (HHDF) and preferring spatial edges over hierarchical edges (HSDF).

To conclude the detailed examination of the different methods in terms of workload balance, the influence of multiple diffusion steps per timestep is studied. Figure 5.9 shows the standard deviation using one to four steps of the HD repartitioning scheme per timestep. More steps in generally lead to a slightly better balance with the highest improvement achieved when going from one to two steps. Even more steps have a slight advantage during the first third of the runtime, while the system establishes its' initial balance. Afterwards, the difference between two or more steps is negligible. The best improvement in distribution is achieved at times $t = 97$ and $t = 253$, reducing the peaks in standard deviation by one. Figure 5.10 illustrates the distribution of process to grids at time $t = 97$ and four iterations of the method per timestep. The overall distribution is similar to before with almost identical percentages of process within one and between one and two standard deviations from the mean. However, the amount of process which have not yet participated in balancing has been reduced from 32% to 25%.

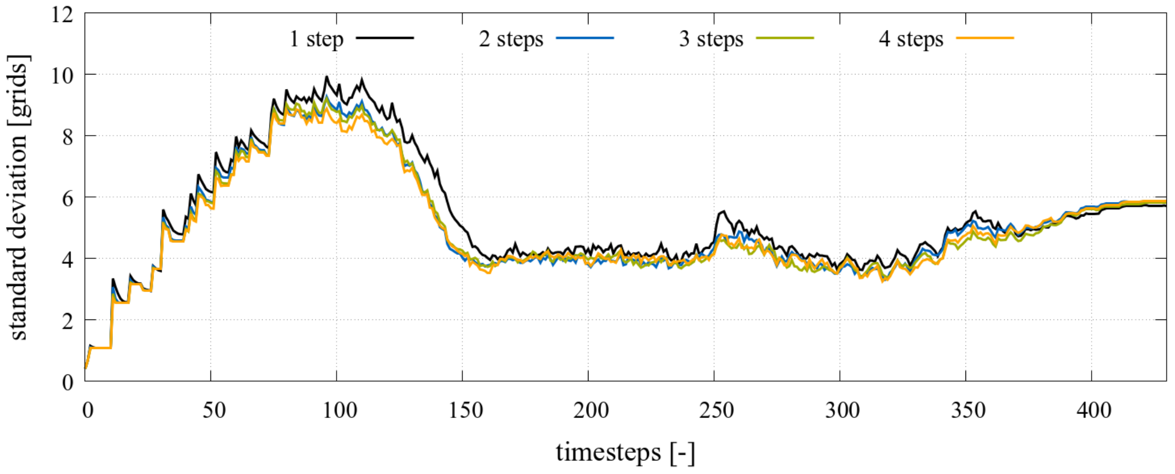


Figure 5.9: Standard deviation of grids per process measured at every timestep for different amounts of iterations of the diffusion repartitioning with highest degree target determination (HD).

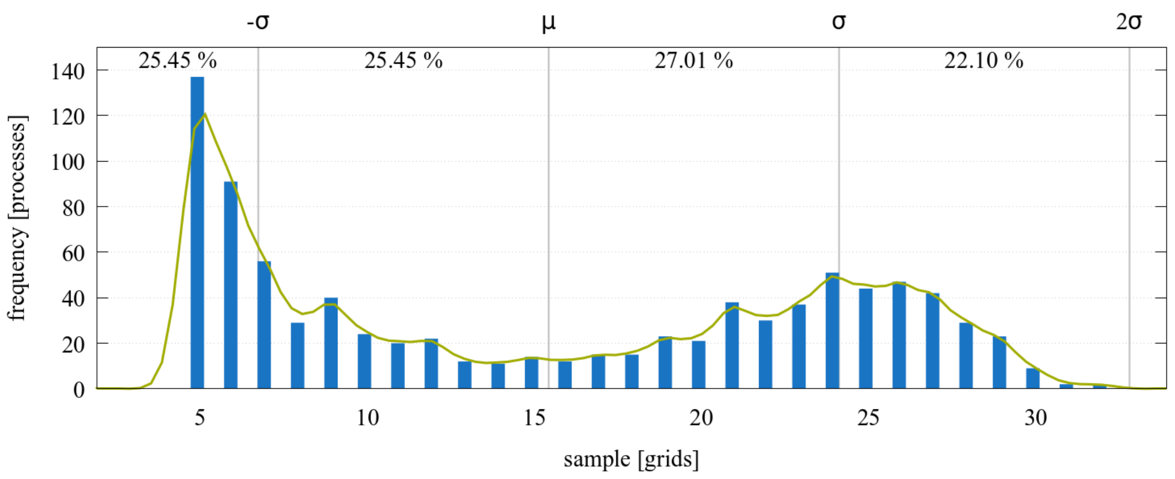


Figure 5.10: Histogram of the frequency distribution of grids per process at time $t = 97$ with four iterations of the diffusion partitioning and highest overall degree target determination (HD). The histogram is overlaid with the smoothed kernel density estimate using a Gaussian kernel.

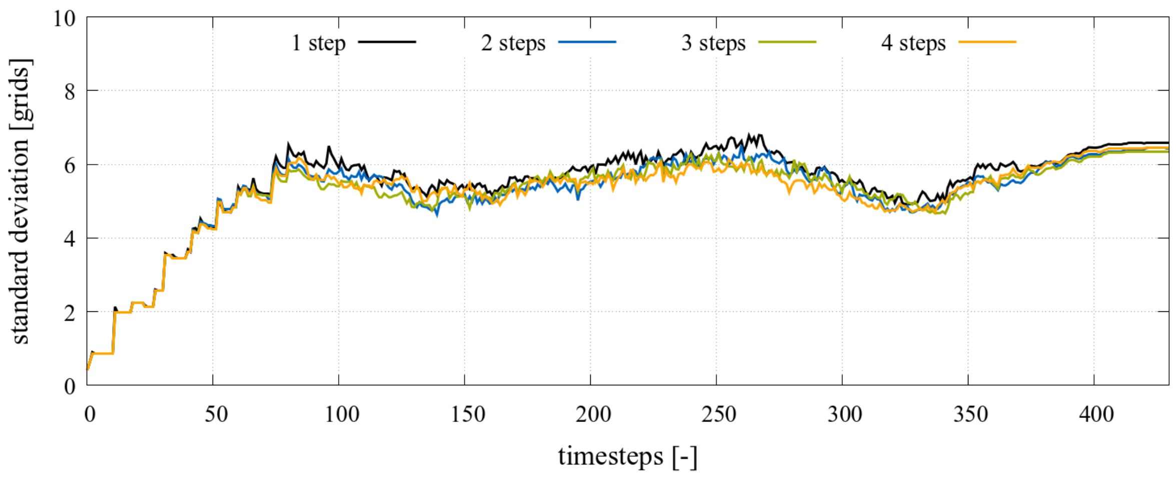


Figure 5.11: Standard deviation of grids per process measured at every timestep for different amounts of iterations of the diffusion repartitioning with highest degree target determination and forced migration (HDF).

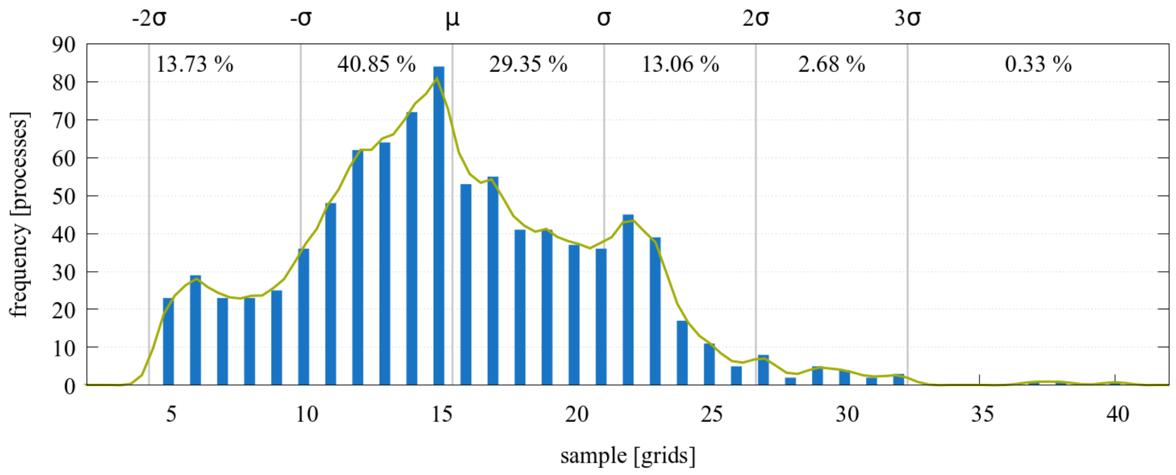


Figure 5.12: Histogram of the frequency distribution of grids per process at time $t = 97$ with four iterations of the diffusion partitioning and highest overall degree target determination with forced migration (HDF). The histogram is overlaid with the smoothed kernel density estimate using a Gaussian kernel.

Similar observations can be made for the target determination method with forced migration HDF. In Figure 5.11 the standard deviation for one to four steps of the HDF scheme is illustrated. Even though peaks in standard deviation are reduced as well, the effect is even less pronounced. The best effects are achieved at times $t = 97$ and $t = 267$, going from a single step to four balancing steps with reductions in standard deviation of roughly one. In Figure 5.12 the distribution of grids per process is depicted at time $t = 97$ and four balancing steps per timestep. Here, the improvement is more noticeable. 70% of processes lie within one standard deviation from the mean compared to 61% with a single balancing step. The percentage of processes which have not yet participated is more than halved, from 12.5% to 5.8%

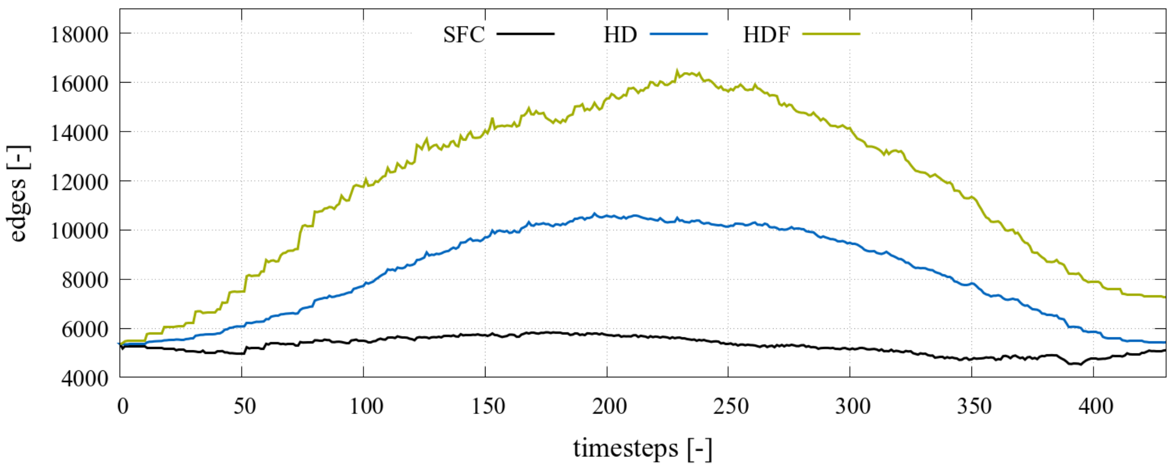
5.1.3 Connectivity

The second key measure observed is the connectivity within the domain. The connectivity can be split into two categories. First the process connectivity. It measures the number of connections (edges) between processes that hold one or more neighbouring grids and are considered neighbouring processes. As such, the process connectivity gives a measure of the communication frequency. A lower number of edges directly translates to a lower number of messages send. Figure 5.13 illustrates process connectivity in terms of total edges during every timestep of the simulation. In Figure 5.13a an overview is given, comparing the SFC strategy against the diffusion methods with highest degree target determination with and without forced migration (HD and HDF). As expected, the SFC performs best out of the three, with a total edge count between 5,834 and 4,523 over the course of the runtime. The process connectivity for both diffusion strategies rises up to a maximum of 10,670 edges at $t = 195$ for HD and 16,458 at $t = 229$ for HDF. Consequently, at its highest, HD has about twice as many messages to send during each process communication cycle and HDF even has more than three times as many as with the SFC repartitioning strategy. HDF performs worst out of the three during the runtime and is also worse off at the end, when both the SFC and HD strategies tend towards their initial state. As the HDF strategy is designed to force migrations even if neighbourhood connections are unfavourable this behaviour is to be expected.

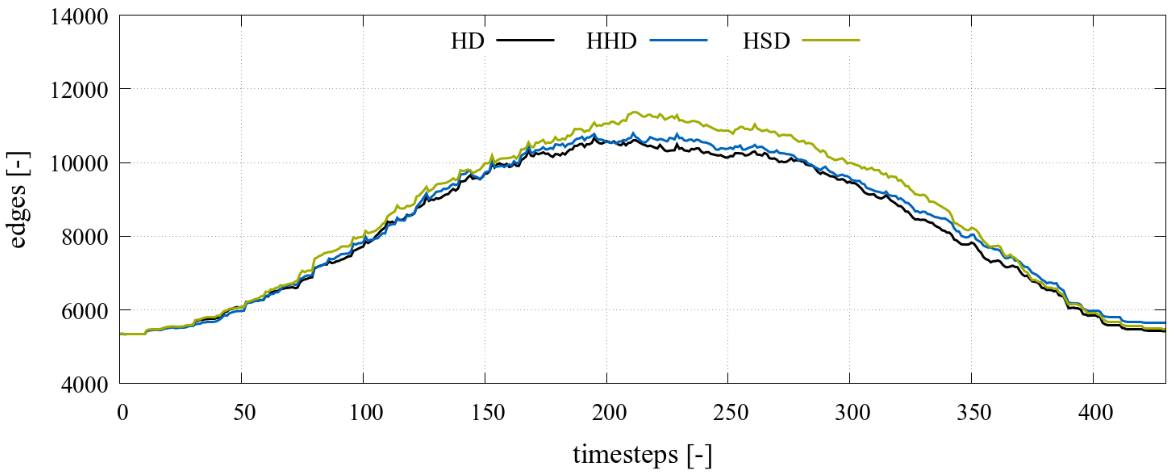
Figure 5.13b gives a closer comparison over the three target determination strategies without forced migration. The already shown HD, as well as the variant which prefers hierarchical edges within the target determination (HHD) and the variant which prefers spatial edges (HSD). The differences are miniscule however. HHD performs slightly worse than HD and reaches its peak at $t = 210$, with 10,793 total edges. An increase of 123 edges compared to HD. HSD performs worst out of the three. Its peak is reached at $t = 212$ with 11,368 total edges.

A similar comparison for the three variants with forced migration, HDF, HSDF and HHDF, is illustrated in Figure 5.13c. The difference between the three variants is even more negligible. The curves rarely deviate from each other, with peaks in total edges of 16,213 at $t = 229$ for HHDF and of 16,721 at $t = 230$ for HSDF.

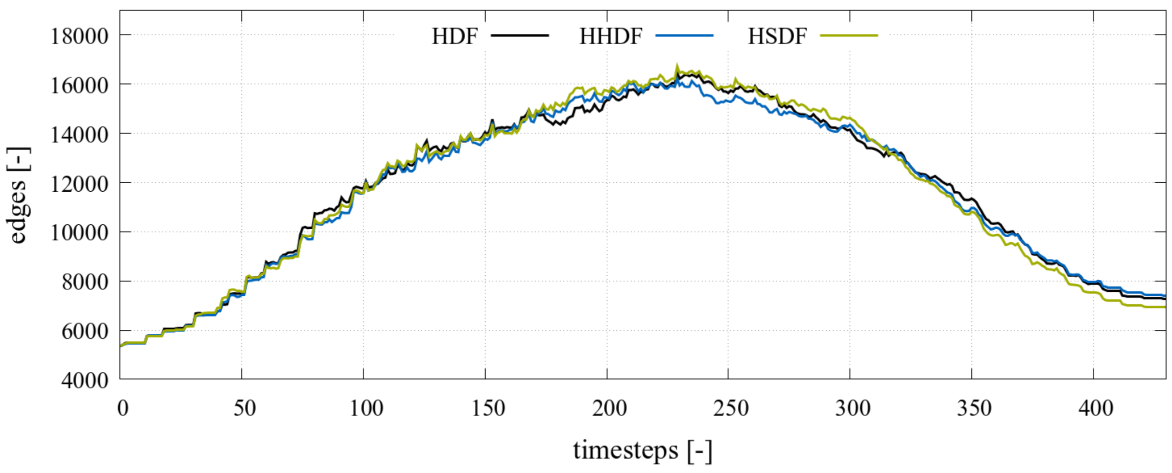
For more insight into the distribution of neighbourhood connections, the frequency distribution of these connections per process is illustrated in Figure 5.14. The first histogram 5.14a shows the distribution of edges per process using an SFC strategy at $t = 175$, where the number of total edges in the domain is at its highest. The mean number of edges per process μ is 13. The standard deviation σ is 5. 72% of all processes lie within one standard deviation from the mean. This means they have between 8 and 18 neighbourhood connections. 22% of processes are between one and two standard deviations, that is processes with less than 8 and between 18 and 23 neighbourhood connections. There are some outliers with more than three standard deviations from the mean. However, in total less than two percent. The highest amount of neighbour connections is 34. The sample with the highest frequency of 122 is 11. In other words, 122 processes have



(a) Overview of the main repartitioning strategies: SFC, diffusion with highest degree target determination (HD) and diffusion with highest degree target determination and forced migration (HDF)



(b) Diffusion repartitioning with highest degree target determination (HD), preference of hierarchical degree (HHD) and preference of spatial degree (HGD).



(c) All diffusion repartitioning strategies with forced migration. Highest degree target determination (HDF), preference of hierarchical degree (HHDF) and preference of spatial degree (HGDF).

Figure 5.13: Process connectivity measured as number of edges between processes in the domain per timestep.

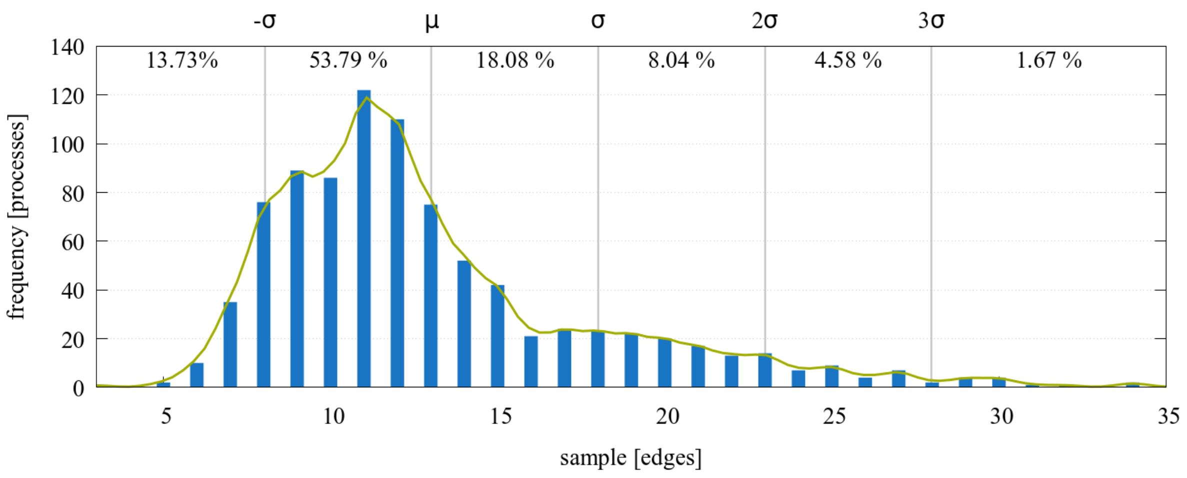
11 neighbourhood connections.

The second histogram 5.14b shows the distribution of edges per process using the HD strategy at $t = 195$. Again, the timestep is chosen at the highest total edge count during the runtime. The mean number of edges per process μ is 24. The standard deviation σ is 10. 78% of all processes lie within one standard deviation from the mean. These processes have between 14 and 34 neighbourhood connections. Compared to the SFC strategy a slightly better clustering around the mean, however the mean and standard deviation are about twice as high. The sample with the highest frequency of 63 is 16. Half as many occurrences as SFC's most frequent sample. 18% of processes are between one and two standard deviations. There are also more outliers that are farther away. The largest outlier is one process with 84 connections. The degree of the process directly influences the workload swapping ratio between neighbouring processes. A high degree leads to a low ratio and, because of the rounding in the diffusion formulation, to few grid transfers. In turn, leading to a high amount of grids, which in turn may cause a high number of neighbourhood connections.

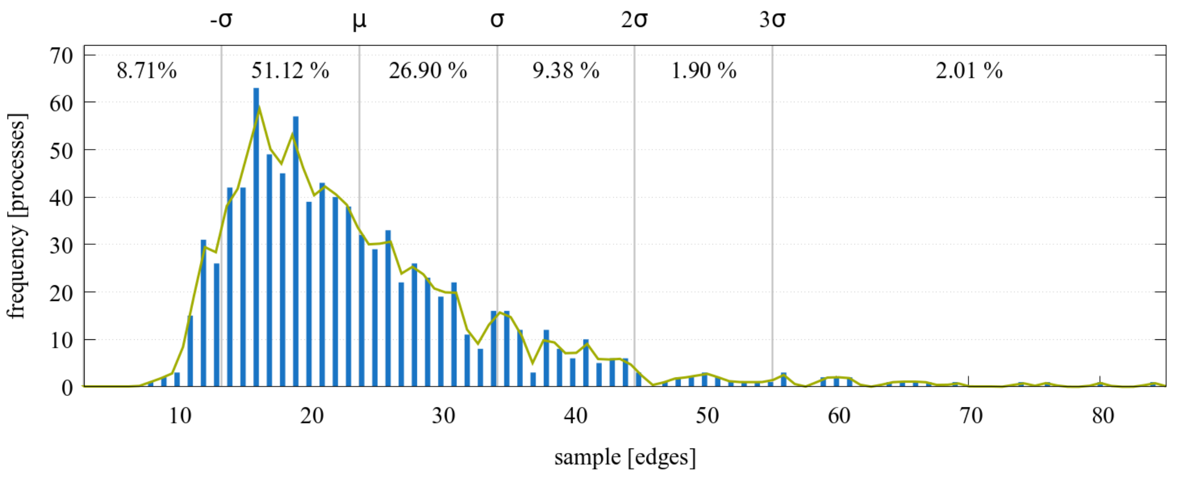
The final histogram 5.14c shows the distribution of edges per process using the HDF strategy at $t = 229$, the respective timestep with the highest total edge count. The mean number of edges per process μ is 37. The standard deviation σ is 18. Mean and standard deviation are around three times as high compared to the SFC distribution, making this the most wide and worst distribution of neighbourhood connections among the three. The overall distribution is however comparable to the former strategies. 73% of processes have between 19 and 55 neighbourhood connections, that is within one standard deviation from the mean. The sample with the highest frequency of 44 is 20. This is again in line with the distribution being roughly three times as wide as the SFC distribution. 22% of processes are between one and two standard deviations, i.e. have less than 19 and between 55 and 73 neighbourhood connections. Outliers with more than three standard deviations from the mean make up 1% of all processes. About half as many compared to the HD strategy. It can be assumed that while this strategy still suffers from a high degree and consequently a low workload swapping ratio, the fact this strategy is not constrained by target determination restrictions somewhat mitigates the tendency towards very high grid amounts. Nevertheless, the process with the highest number of connections has 115. It is connected to roughly 13% of all participating processes.

To complete the picture, the second connectivity category is the grid connectivity. This is measured as the amount of inter-process edges directly between grids. At any given timestep the amount of grids, and consequently the total amount of inter- and intra-process edges is equal. Intra-process edges do not entail any communication, because the necessary values can simply be read or copied from the local memory. Therefore, a lower amount of inter-process edges, which directly translate to communication over the network, is beneficial. In Figure 5.15 the inter-process edges for all implemented repartitioning strategies are illustrated. To be comparable, the values are taken at the same timestep $t = 253$, where the computational intensity is maximal.

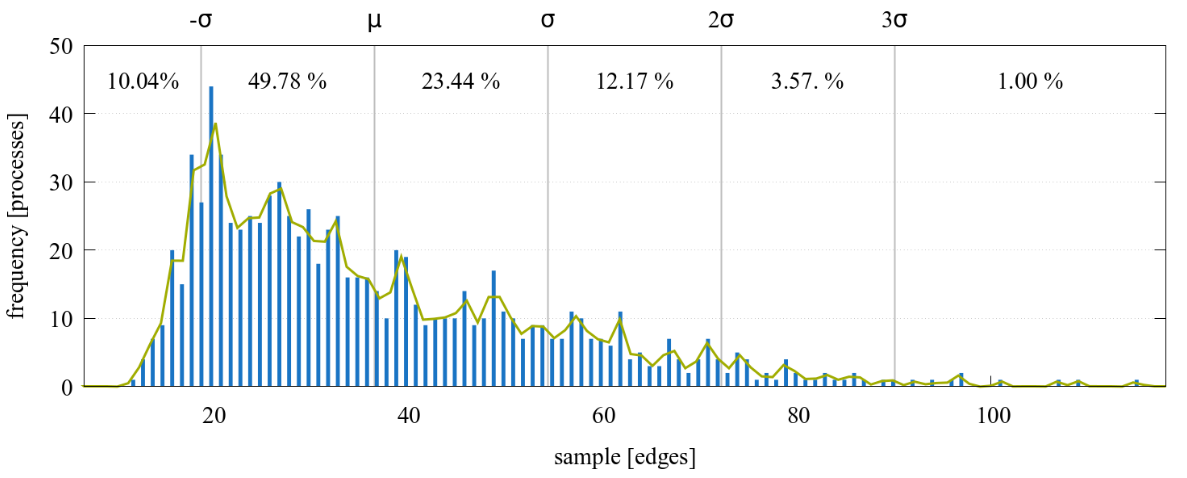
Exhibiting 56,083 inter-process edges, the lowest of all measured strategies, the SFC repartitioning shows the best distribution of neighbouring grids. This means, the clustering



(a) SFC distribution at time $t = 175$.



(b) HD distribution at time $t = 195$.



(c) HDF distribution at time $t = 229$.

Figure 5.14: Histogram of the frequency distribution of neihgbourhood connections (edges) per process. The histogram is overlaid with the smoothed kernel density estimate using a Gaussian kernel.

of neighbouring grids is best and the surface area is indeed minimal, as advertised. Furthermore, the percentage of hierarchical edges compared to spatial edges is the lowest of all strategies, with 14% hierarchical and 86% hierarchical edges. This suggests that, compared to other strategies, the SFC method largely favours hierarchical edges to be kept within the same process over spatial edges. If the SFC strategy is seen as the gold standard, weighting hierarchical edges even more than the HHD and HHDF strategies do, might be worthwhile.

The total amount of inter-process edges of the diffusion strategies are roughly equal. HD exhibits the lowest amount of 79,933, followed by HHD with 83,070, HDF with 84,781, HSDF with 85,237, HHDF with 85,195 and finally HSD with 86,090 inter-process edges.

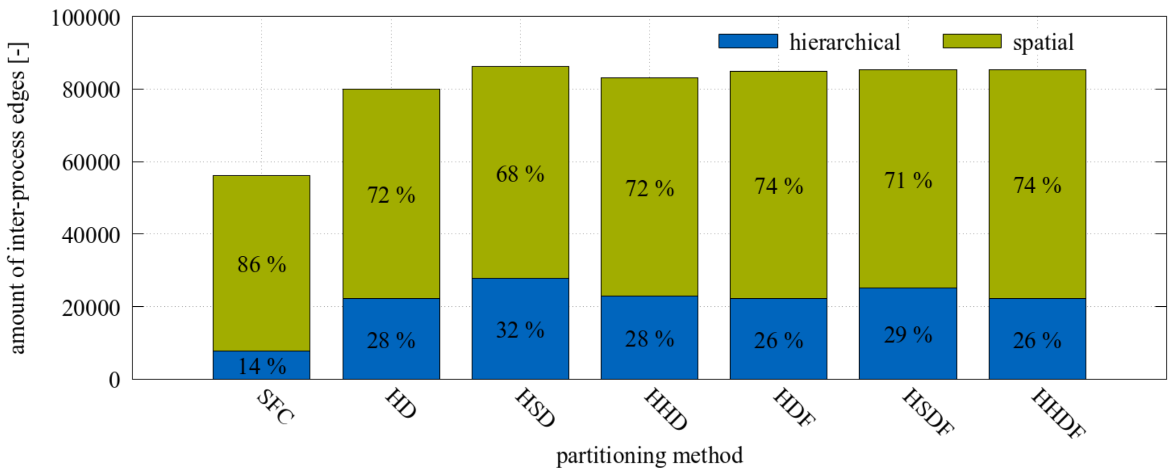
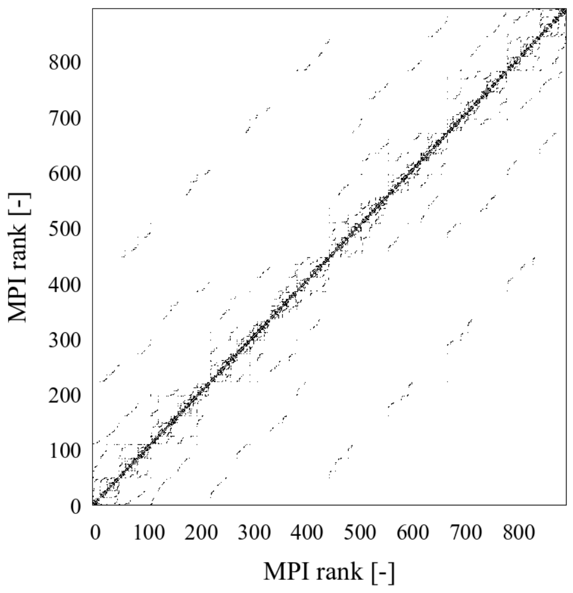


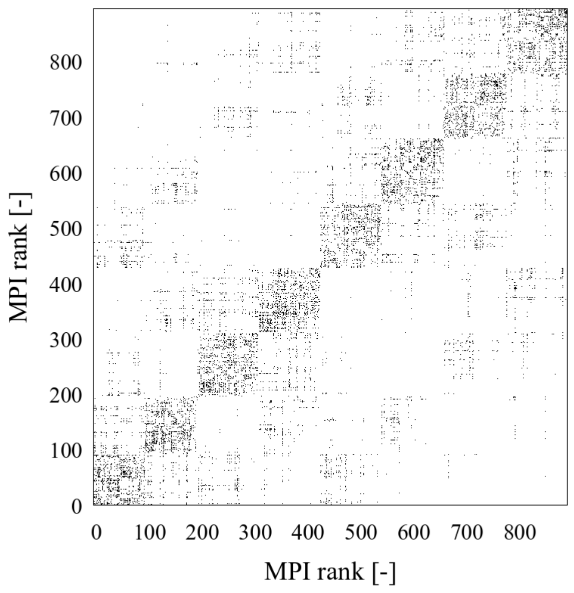
Figure 5.15: Amount of total inter-process edges between grids, divided into spatial and hierarchical edges for all repartitioning strategies at time $t = 253$.

In terms of distribution of hierarchical compared to spatial edges, the overall highest degree strategies show no difference compared with the strategies that explicitly prefer hierarchical edges. However, the versions which prefer spatial edges show a lower percentage of inter-process spatial edges compared to inter-process hierarchical ones. In general, the percentage of inter-process hierarchical edges is lower for the versions with forced migration compared to their non forced counterparts. Given that the overall edge count of all grids is equal for every strategy and the inter-process edges of the diffusion methods are roughly equal as well, the main influence on the higher process connectivity of the versions with forced migration can be attributed to the worse clustering of neighbouring grids onto individual processes.

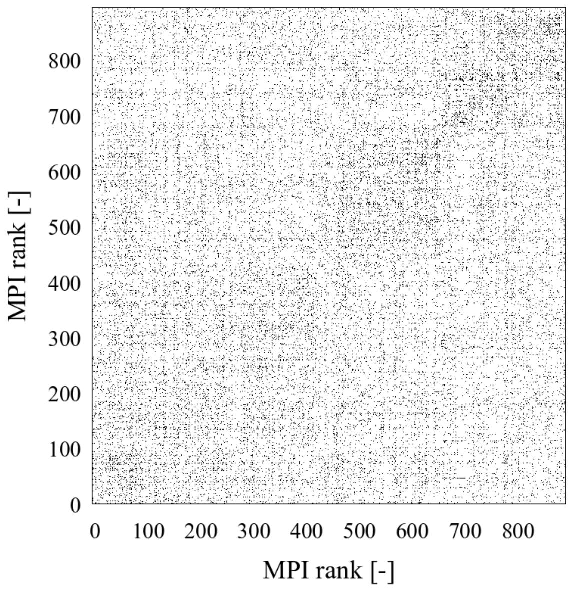
This may also be concluded from Figure 5.16. Here, the process connectivity is depicted by means of a matrix with the respective MPI ranks as rows and columns. If two ranks share a neighbouring connection, the respective matrix entry is marked in black. As a communication link is bidirectional, the matrix is naturally symmetric. One observes a very tight clustering around the main diagonal for the SFC repartitioning method (Figure 5.16a). Numerically close ranks mainly communicate with each other with some exceptions marked by off-diagonal entries. The clustering is still visible for the HD strategy (Figure 5.16b), albeit much wider with many more off-diagonal entries. Because, the strategy cannot transfer grids with no neighbourhood connections to a target rank, the deviation from the initial SFC clustering is limited. When using the HDF strategy (Figure 5.16c), almost no clustering is visible anymore. One might surmise a faint remnant of the initial SFC pattern around the main diagonal, however hardly visible.



(a) SFC



(b) HD



(c) HDF

Figure 5.16: Process connectivity at time $t = 253$.

5.1.4 Migration

To reach the computed partitioning, the respective grids must be transferred from one process to the next. As such, to conclude the observations of the repartitioning strategy, these redistribution costs are evaluated. The main findings comparing the SFC, HD and HDF strategies are illustrated in Figure 5.17 and Table 5.1. Results for the other strategies are purposely omitted, because within the variants with and without forced migration, the differences are negligible.

Here, the main drawback of the otherwise advantageous SFC repartitioning becomes visible. It incurs massive redistribution costs. In total, grids are migrated 7.3 million times. Almost every grid (99.99%) is migrated during its lifetime, on average 24 times. During the 430 timesteps of the testcase, the highest number a single grid is migrated is 302 times. To reach the state computed by the method, 16,935 migrations are necessary per timestep on average. The highest migration count per timestep is 49,289 at time $t = 252$.

The HD and HDF methods incur only a fraction of the redistribution costs. HD accumulates 250 thousand migrations in total, only 3.4% of the migrations of SFC. HDF accumulates even less, with 200 thousand migrations in total, which is 2.7% of the migrations of SFC. While SFC migrates almost all grids during their lifetimes multiple times, less than half of the grids are migrated even once during their lifetime using the diffusion strategies. 45% using HD and 40% using HDF. For HD, on average 586 grids are migrated per timestep. The highest single number of migrations is 2,066 at timestep 252. For HDF, the highest number of migrations is measured at the same timestep, with 1,735 migrations. The average number is 456 migrations per timestep.

When comparing HD and HDF, it can be observed that up to around time $t = 25$ HDF leads to more migrations. This is in line with expectations. However, afterwards HD overtakes and performs about 25% more migrations in total. At first glance this might seem counter intuitive. Nevertheless, this is in line with the observed better workload balance from the second third of the runtime onwards.

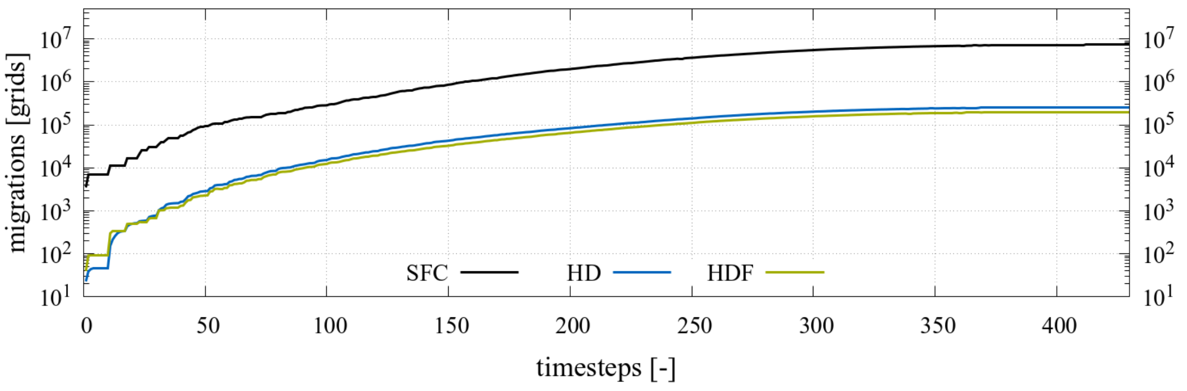


Figure 5.17: Total amount of migrations per timestep for SFC, HD and HDF strategies.

The impact on the runtime of the added migrations of the SFC method is significant. As no actual calculation takes place, the majority of the runtime comes from evaluating the

Table 5.1: Migration statistics for the growing sphere test example.

	SFC	HD	HDF
total migrations	$7,316,002$	$253,087$	$196,926$
individual grids migrated	$294,874$	$134,326$	$118,450$
ratio [%]	99.99	44.84	39.54
average migrations per step	$16,935$	586	456
max migrations per step	$49,289$	$2,066$	$1,735$
average migrations per grid	24.42	1.88	1.66
max migrations per grid	302	20	15

distribution and the subsequent grid transfers. Multiple runs with the same conditions were performed. On average the diffusion approach with HD target determination takes 73 sec or 1:13 min. The SFC strategy needs more than 33 min on average for one complete run. This is in line with the roughly 33 times higher migration volume of the later method.

5.1.5 Short Summary

This test example analysed a pure artificial AMR case to test the repartitioning. During the course of the runtime, the complete domain has been refined to depth $d = 6$ once and coarsened back to $d = 4$. This example aimed to give an overview over all available target determination methods of the diffusion repartitioning strategy as well as to analyse the influence of a greater number of iterations of the method.

In terms of workload balance, the diffusion model for repartitioning with any target determination method performs reasonably well when it comes to establishing a balanced workload among all participating processes. After relatively high standard deviations at the beginning of the simulation, when the number of grids grows rapidly, the various diffusion methods with and without forced migration tend towards a relative standard deviation of less than 10 percent. With more time, the distribution of grids per process tend to become more narrow with less outliers. Comparing the forced and the non forced variants, it becomes clear that to establish an initial system balance, the forced variants have advantages, because they are less limited. After an initial balance has been established however, and all processes have participated in grid migrations, the non forced variants outperform the forced variants slightly. The distribution of all variants can be improved by using more balancing steps per timestep, albeit minor. The best improvements could be observed going from one balancing steps to two. Afterwards the returns are diminishing.

Looking at process connectivity, the diffusion variants without forced migration end up with around twice as many communication links at their respective maxima compared to the SFC method. The versions with forced migration perform worse as expected, with around three times as many communication links. This means every communication cycle is two, (or three) times as expensive. Furthermore, the overall distribution is worse, with a standard deviation twice as high for the non forced migration versions compared to the SFC baseline and three times as high for the forced migration versions. Concerning the overall inter-process grid connectivity, all diffusion-based approaches have around 1.5 to 1.6 times as many inter-process edges as the SFC method, whereas the later shows a preponderance for spatial edges. Even though the amounts of edges for the diffusion approaches are roughly equal, the clustering is worse for the forced migration methods. This is one of the main reasons responsible for the worse process connectivity.

Even though the SFC method is incremental, meaning small changes in the refinement structure only lead to small changes in the partitioning, with most of the grids still assigned to their former processes, the number of grid migrations is considerably higher compared to the diffusion approach. The combined number of migrations of the SFC is about 33 times that of the diffusion repartitioning. As this is a pure AMR benchmark without any computation, the impact of the higher migration volume on the runtime is considerable. Using the SFC method the time is around 27 times as long as with the diffusion method with highest degree target determination.

5.2 Test Example - Additive Manufacturing Benchmark

The second example is inspired by an additive manufacturing benchmark test: a laser powder bed fusion on bare metal substrates. The AMB2018-02 benchmark serves as a starting point for the evaluation [198]. In these benchmark cases, a laser tracks over a bare nickel-based superalloy metal surface in individual strokes, melting the surface of the alloy. For a physically correct simulation, the interested reader is referred to Kopp et. al. [187]. The present test setup is inspired by their hatched square laser track. As for the present evaluation, physical correctness is less important. Therefore, the model is simplified to a three-dimensional temperature diffusion equation. This does not capture the phase change of melting substrate for example.

5.2.1 Test Setup

The simplified physical model is represented by the three-dimensional temperature diffusion equation

$$\frac{\partial}{\partial t}T = \alpha \cdot \Delta T. \tag{5.1}$$

with T being the temperature, depending on time t and its location in the grid. α represents the thermal diffusivity in [m^2/s], Δ is the Laplace operator. For the numerical discretisation a central difference scheme in space and a forward Euler timestepping is used. This is known as FTCS scheme, which is commonly used to solve parabolic partial differential equations.

The domain setup consists of a cubic alloy of size 25 cm $\times$ 25 cm $\times$ 5 cm. The root node, representing the complete domain, is refined by subdividing the domain in x- and y-direction by two. There is no subdivision in z-direction. Consequently, the root node has four children on refinement depth $d = 1$. Refinements from level $d = 1$ onwards use a subdivision of two in all three directions, resulting again in an octree structure with eight children per refined parent node. The domain is uniformly refined to depth $d = 5$, which results in 18,725 total nodes and 16,384 leaf nodes on refinement level $d = 5$.

The initial partitioning follows the domain generation method outlined in section 3.4. Therefore, all test cases start from a Z-order SFC partitioning. Within the domain generation procedure, each node is discretised with a grid, which consists of $16 \times 16 \times 8$ cells. Each cell is initialised with an initial temperature of $T = 25°C$. The material for the chosen alloy is steel with a thermal diffusivity of $\alpha = 0.208 \cdot 10^{-4}$ m^2/s. The laser is represented by a time-dependent Dirichlet temperature boundary condition applied to the top surface of the alloy, with a temperature of $T = 1,200°C$. The radius of the laser is 750 μm.

The laser travels along a prescribed path with a speed of 80 mm/s. Initially, it draws a square on top of the alloy with a side length of 15 cm, starting from the square's right lower corner and moving in y-direction first. The square is completed and the initial position is reached again at time $t = 7.5$ s. After completing the square, the laser hatches the area inside the square, starting again from the initial position at its right lower corner. The spacing between each track of the hatch is 5 mm. The hatch is complete around time $t = 61$ s. Figure 5.18 illustrates the accumulation of the laser track at four different timesteps. In practice, the temperature boundary condition is removed after the laser has moved away from the respective cell.

In regions where the temperature diffusion is high, in the vicinity and in the wake of the laser, an increased accuracy of the solution is desired. Thus, grids are refined incrementally to depth $d = 7$ when the second derivative of the temperature exceeds a value of $260°C$ in one of its cells. Consequently, an incremental coarsening is performed back down to the initial depth of $d = 5$ when a threshold of $30°C$ is undershot. Refinements and coarsenings need to satisfy the 2:1 balance condition as before. Figure 5.19 serves to illustrate the refinement state at time $t = 25$ s. The left upper illustration shows the complete domain. Cells with a temperature of greater than $30°C$ are coloured. Below is a zoomed in view of all grids on depths $d = 6$ and $d = 7$. The current laser spot is clearly visible in red as well as its previous path where temperature and consequently the diffusion are still high and the domain is refined.

The timestep size is determined by a stability analysis of the FTCS scheme. It takes into account the cell size and the thermal diffusivity and results in $dt = 9 \cdot 10^{-5}$ s. The laser speed is taken into account as well, which limits the timestep such that the laser cannot travel more than the length of a single cell per step. However, the limiting factor for dt is the stability condition. A domain update is performed every 20 timesteps, or every $dt = 18 \cdot 10^{-4}$ s. After every domain update, four repartitioning steps of the diffusion method are performed to balance the workload.

Figure 5.20 shows the evolution of the number of grids during the runtime of the test. At time $t = 0$ s the domain has 18,797 grids with roughly 38.5 million cells in total. The number of degrees of freedom correspond to the number of cells. The initial refinement around the laser spot increases the grid count to 19,053 at time $t = 0.54$ s, with roughly

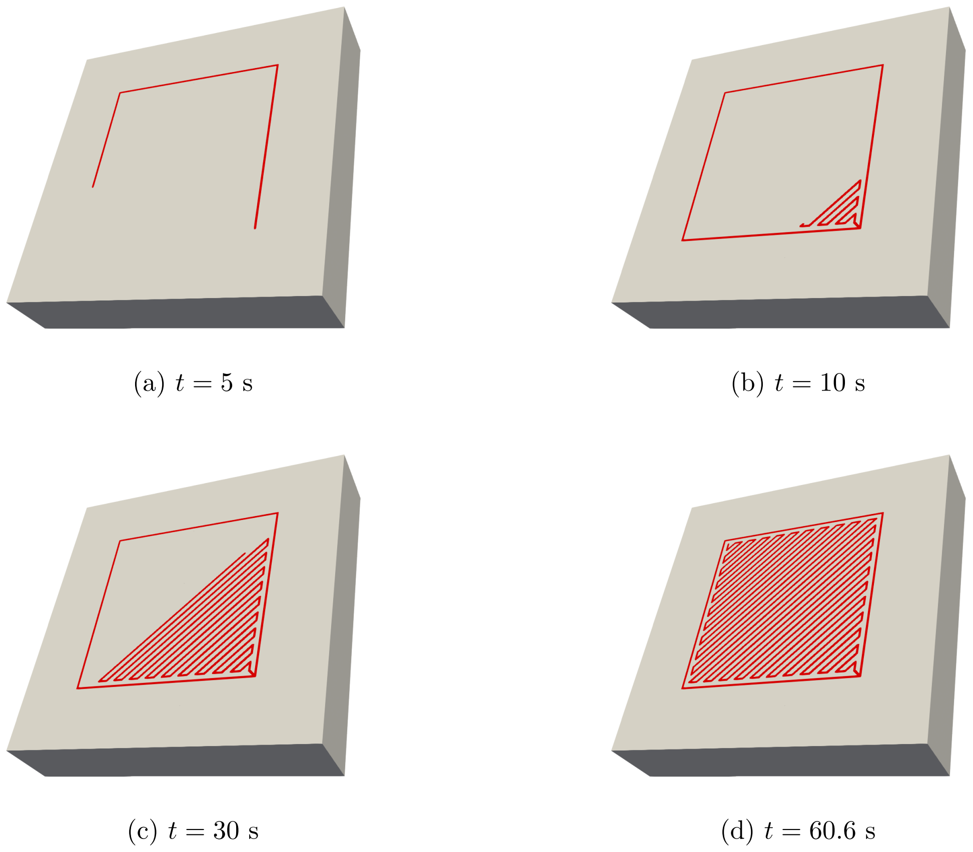


Figure 5.18: Accumulation of the laser track boundary condition on top of the alloy at different times.

39 million cells. The amount of grids stays relatively constant until time 7.5 s, at which point the initial square is completed. Small peaks can be observed when the laser changes direction in the corners of the square. During the hatching, the grid count shows a greater variability between 19,013 and 19,197, with between 38.9 and 39.3 million cells. However, the total variability is less than 2%. During the course of the test case 90,453 unique grids have existed in total.

Compared to the first example, this test examines very small localised phenomena with residual effects. The refinement and coarsening of the domain closely follows the laser track. In the following, the diffusion-based repartitioning with highest degree target determination (HD) is examined in terms of workload balance, connectivity and migration frequency. A Z-order SFC repartitioning method serves as a baseline. To allow an evaluation of the influence of the number of participating processes on the repartitioning, this testcase has been deployed to 4, 8, 16 and 32 nodes of the CoolMUC-2 cluster, with 28 cores each. Again, each core runs a single process, resulting in 112, 228, 448 and 896 processes in total.

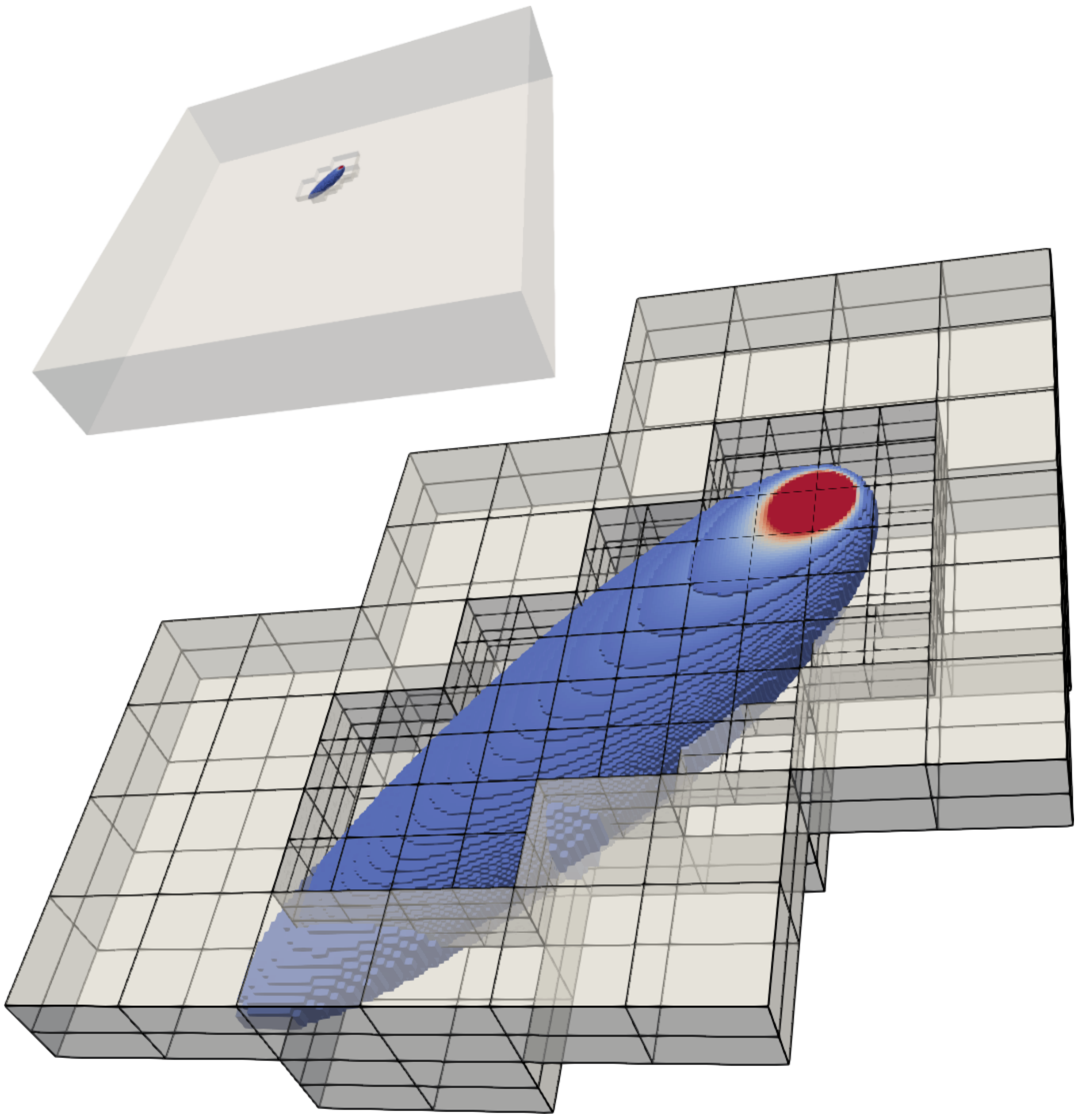


Figure 5.19: Refinement state at time $t = 25$ s. Cells with temperatures higher than 30°C are coloured. Detailed view is illustrated in the center, complete domain is at the top left.

5.2.2 Workload Balance

Figure 5.21 shows the standard deviation σ of grids per process for the HD repartitioning for the different node counts, from 4 to 32 nodes. It can be seen that with higher amounts of processes, σ becomes less volatile and generally smaller. Using 4 nodes, σ varies between 1.5 and 4.1. Using 8 nodes, the variability is lowered to values of σ between 1.8 and 3.3. Again doubling the node count to 16, σ varies between 1.4 and 2.8 and finally, using 32 nodes, σ varies between 1 and 2.3. This behaviour was to be expected. As the number of processes increase, the mean number of grids per process decreases and the standard

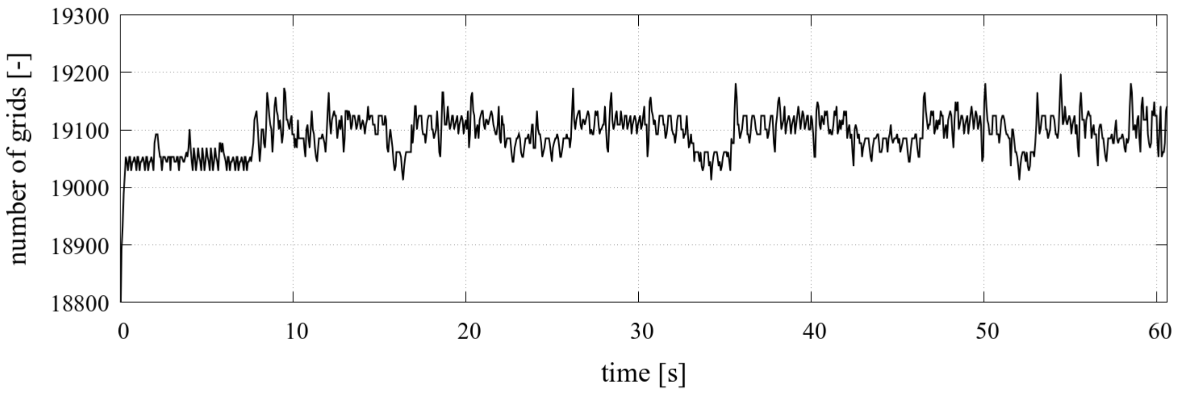


Figure 5.20: Number of grids in the domain per timestep. Data points are sampled for better visibility. Every fourth data point is taken into account.

deviation should become smaller.

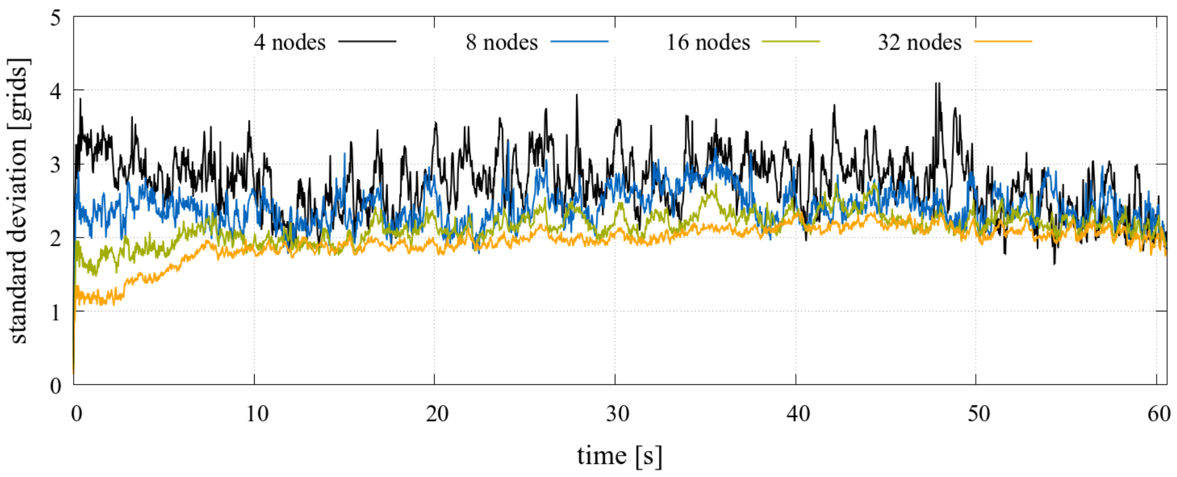


Figure 5.21: Standard deviation of grids per process measured at every timestep using HD diffusion repartitioning and different amounts of participating processes.

The relative standard deviation shown in Figure 5.22 as the ratio between σ and the mean number of grids μ reveals a decrease in distribution quality when using higher node counts. Compared to the absolute σ, the relative standard deviation and its volatility increases with higher number of participating processes. Especially for 32 nodes, the initial square outline of the laser up to time $t = 7.5$ s can be observed clearly by the increase in relative σ from around 6% to 9%. The square outline covers a relatively large domain, making it hard for the diffusion approach to keep up, as each process only covers a small amount of the complete domain. Less processes, each covering a larger amount of the domain do not exhibit this large increase in relative σ. After the initial square has been completed, all curves stay relatively constant with values between 8 and 11 percent for 32 nodes, between 4 and 6 percent for 16 nodes, between 2 and 4 percent for 8 nodes and around 2 percent for 4 nodes.

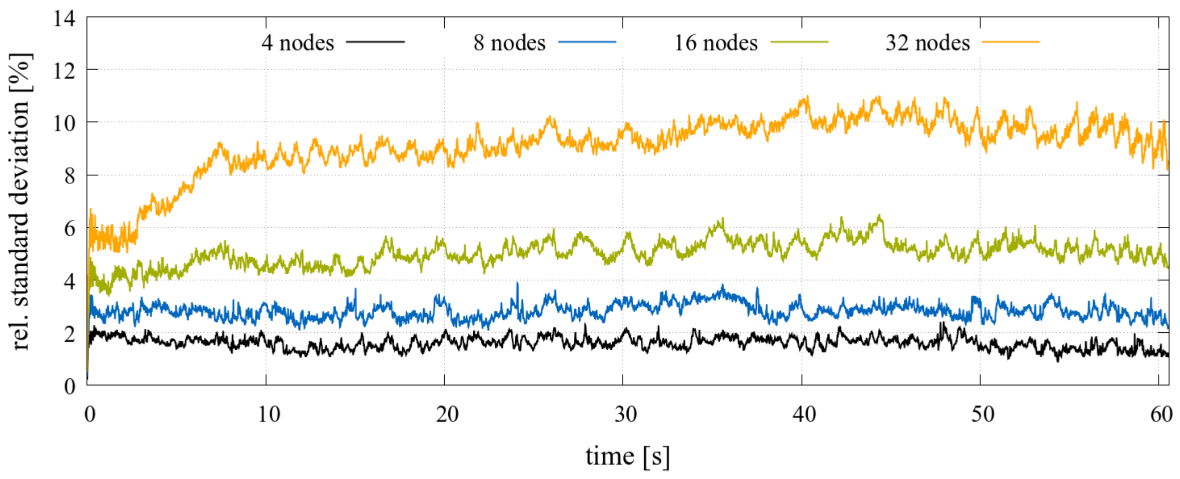


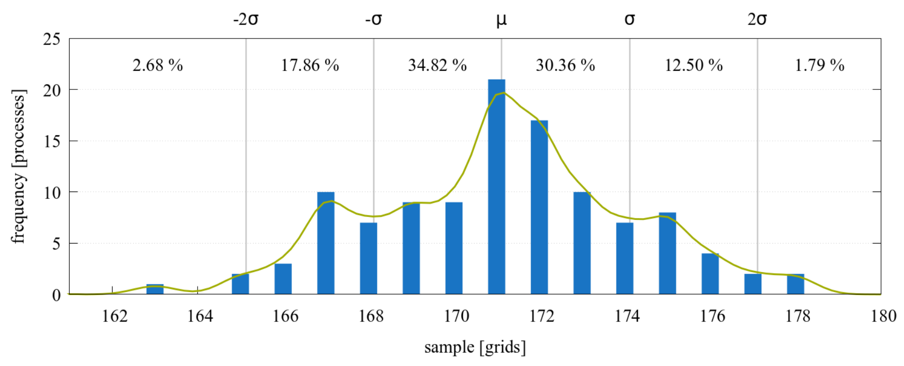
Figure 5.22: Relative standard deviation of grids per process measured at every timestep using HD diffusion repartitioning and different amounts of participating processes.

A complete picture of the distribution at time $t = 38$ s, when the most number of grids exists in the domain, is depicted in Figure 5.23. It can be assumed that for higher node counts, the amount of processes that actively participated in load balancing becomes less. The reason is a combination of very few grids per process and a tendency to keep grids when the weight difference between neighbouring processes is small because of the rounding. Nevertheless, the diffusion method in this case is able to limit the impact of outliers and keep a relatively narrow distribution. Using 32 nodes, the average amount of grids is approximately 21. A single refinement introduces eight new grids, which increases the load by almost 50%. Multiple refinements therefore may increase the current load by a multiple. However, in total there are only five processes with a grid count larger than three standard deviations from the mean (27 grids), among them a single highest sample of 29. On the other side, there are 15 processes with grid counts smaller than three standard deviations from the mean (15 grids). The smallest sample here is a single process with 11 grids. These can be assumed to be the result of an overcompensation in the diffusion process.

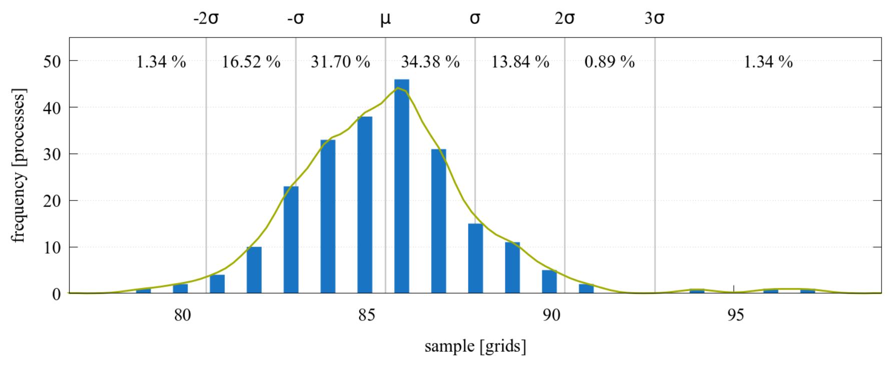
5.2.3 Connectivity

The examination of the domain connectivity is again split into the process and grid connectivity. Figure 5.24 depicts the total number of connections (edges) between processes per timestep for the HD and SFC repartitioning strategies and the different numbers of nodes used. The total connectivity for the SFC strategies stays constant at 560 edges for 4 nodes, 1,270 edges for 8 nodes, 2,640 edges for 16 nodes and 5,110 edges for 32 nodes. Doubling the amount of nodes approximately doubles the amount of total edges in the system, which is reasonable behaviour.

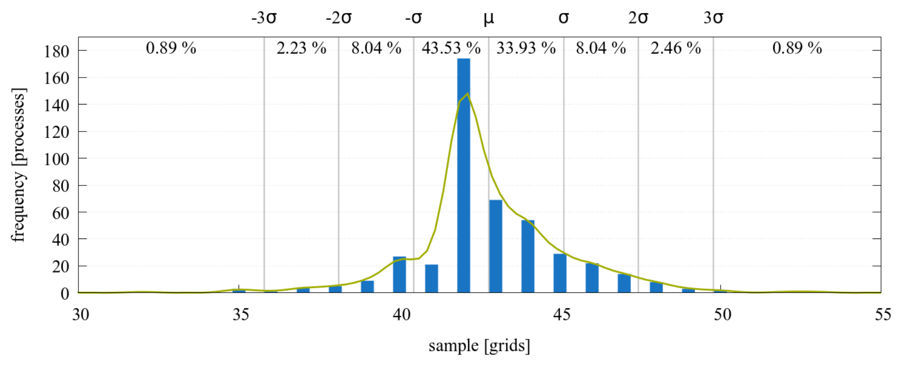
When using the HD strategies the amount of edges increase from their initial amount to time 7.5 s. Afterwards they stay roughly constant until the end of the runtime. Again, this behaviour suggests that similar to the behaviour observed for the workload balance,



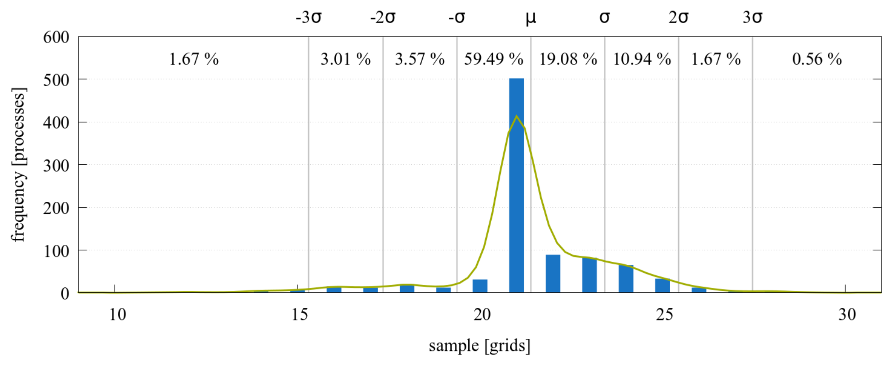
(a) 4 nodes



(b) 8 nodes



(c) 16 nodes



(d) 32 nodes

Figure 5.23: Histogram of the frequency distribution of grids per process at time $t = 38$ s with HD diffusion repartitioning for different node counts. The histogram is overlaid with the smoothed kernel density estimate using a Gaussian kernel.

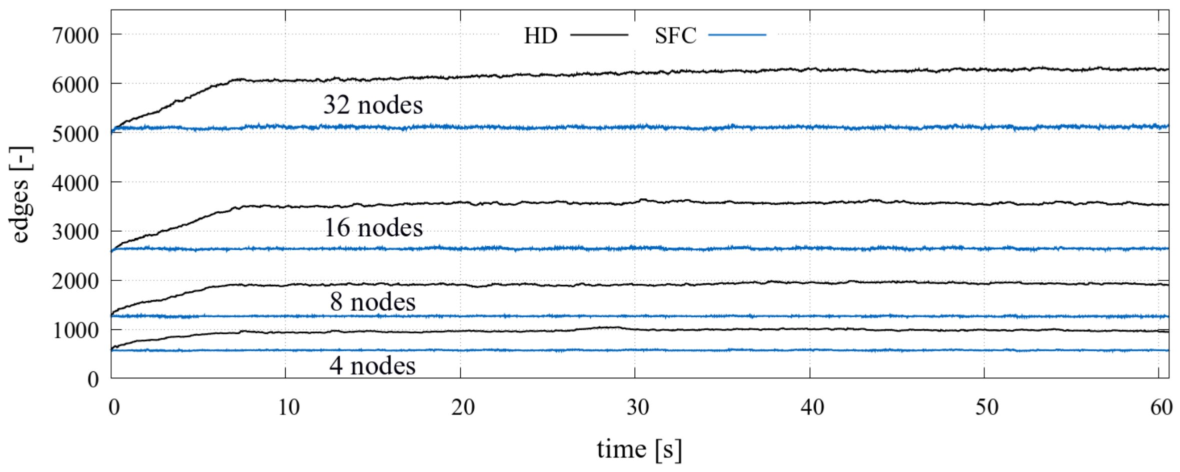


Figure 5.24: Process connectivity measured as number of edges between processes in the domain per timestep. Illustrated are HD and SFC repartitioning strategies with 4, 8, 16 and 32 nodes.

the diffusion strategy has issues with high spatial distances when it comes to process connectivity, too. Small local phenomena are handled well though. Using 4 nodes, the total amount of connections rises to 950 edges. An increase of 70% compared to the SFC method. Using 8 nodes, the edge count rises to 1,900, a 50% increase. For 16 nodes used, the edge count increases by 33% to 3,500 edges. Finally, with 32 nodes the edge count increases by 23% to 6,300. The percentage increase in process connectivity is reduced for a larger number of nodes.

To get a better insight into the quality of the distribution, Figure 5.25 illustrates the standard deviation σ per timestep of both strategies for the different node counts. σ stays almost constant when using the SFC repartitioning at values around 4. Increasing the node count slightly increases the standard deviation, too. The relative σ ranges between 33% and 38%. When using the HD repartitioning, standard deviations are generally higher. Again, the increase up to time $t = 7.5$ can be seen clearly. While at 32 nodes σ stays relatively constant, using less nodes shows a more oscillatory behaviour. Using 4 nodes, σ even increases sharply directly at the beginning. Here, less processes result in a higher connectivity and a larger impact on many processes, when the initial square is drawn by the laser track. Using more processes, the phenomena stay more localised. The relative standard deviation for node counts 8, 16 and 32 increases from 35% to around 45% at time $t = 45$ s, where it remains. The lower node counts show a higher volatility though. A node count of 4 exhibits an initial spike up to 70% and gradually decreases down to 45% as well.

Figures 5.26 and 5.27 illustrate the comparison between the distribution of edges to individual processes for the HD and SFC strategies at time $t = 38$ s, where the computational intensity is maximal. The first comparison used 4 nodes. The SFC strategy exhibits a mean number of edges per process μ of 10.4 and standard deviation σ of 4. 72% of all processes have an amount of edges within one standard deviation from the mean, 95%

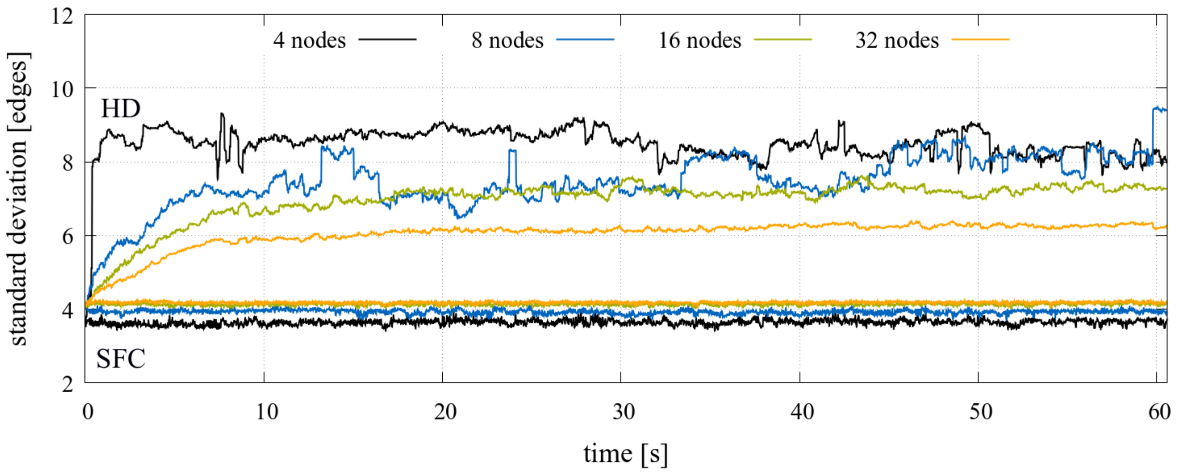
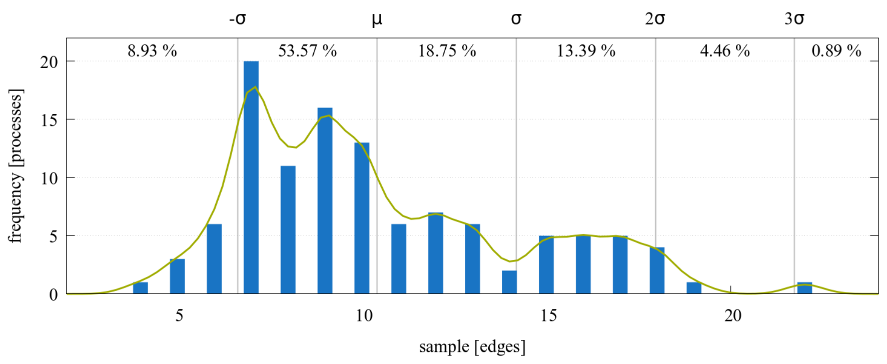
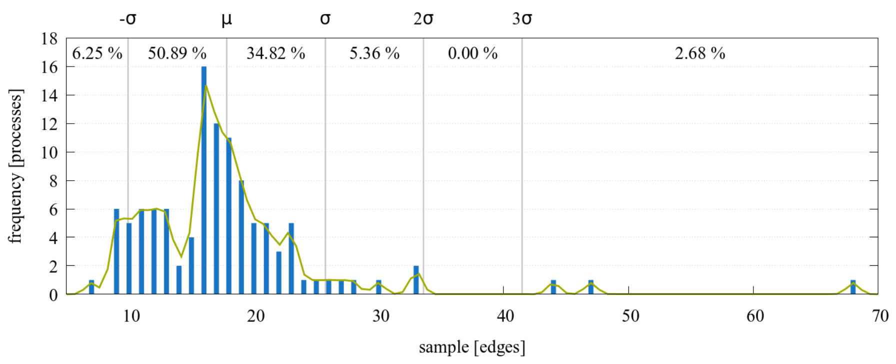


Figure 5.25: Standard deviation of connections per process measured at every timestep using the HD and SFC repartitioning strategies for different amounts of participating processes.

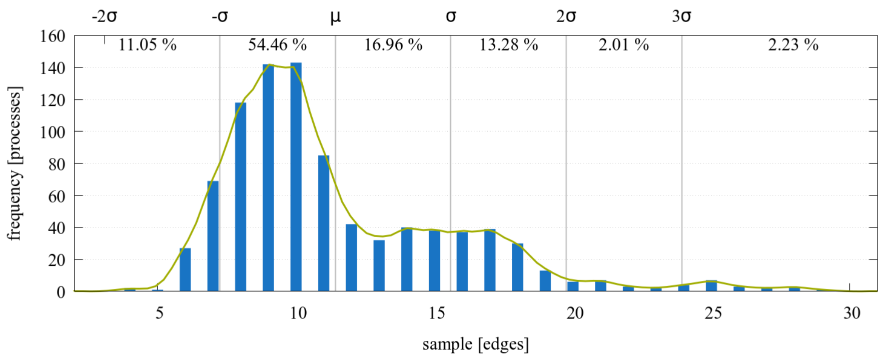


(a) SFC, 4 nodes

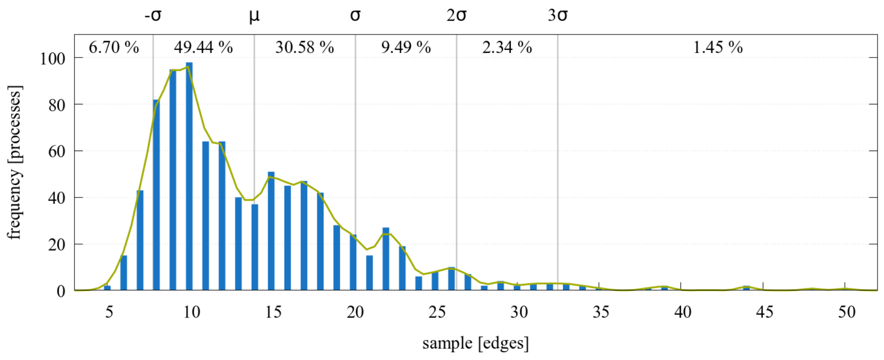


(b) HD, 4 nodes

Figure 5.26: Histograms of the frequency distribution of edges per process at time $t = 38$ s with SFC and HD repartitioning strategies. The histogram is overlaid with the smoothed kernel density estimate using a Gaussian kernel.



(a) SFC, 32 nodes



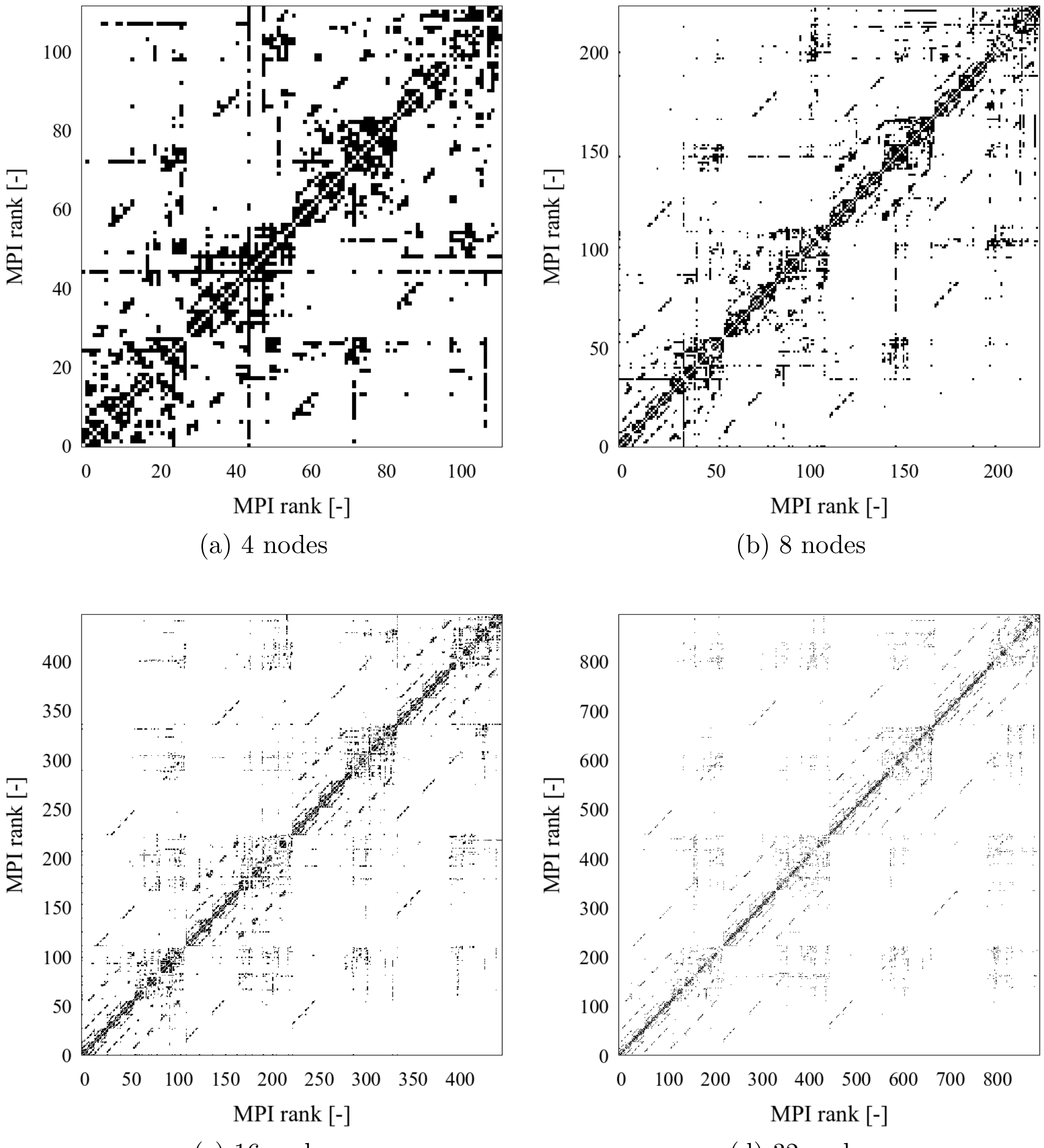
(b) HD, 32 nodes

Figure 5.27: Histograms of the frequency distribution of edges per process at time $t = 38$ s with SFC and HD repartitioning strategies. The histogram is overlaid with the smoothed kernel density estimate using a Gaussian kernel.

lie within two. There is a single outlier with 22 edges. The sample with the highest frequency of 20 is 6. The HD strategy exhibits a mean number of edges per process μ of 18 and standard deviation σ of 8. The distribution is relatively more narrow with 86% of all processes within one standard deviation from the mean and 97% within two. There are more outliers tough. Three processes hold more edges than three standard deviations from the mean. The highest edge count is 68.

The second comparison used 32 nodes. The mean number of connections between processes μ is 11.4, the standard deviation σ is 4.1. The distribution in terms of percentages is very close to the first comparison, with 71% of all processes within one standard deviation from the mean and 96% within two. However, the number of outliers is considerably higher. In total, 20 processes or roughly 2.2% have a number of connections larger than three standard deviations from the mean. The greatest outlier has 29 edges. With the HD method, the mean μ is 14 and the standard deviation σ is 6.2. The distribution differs a little from before, with 80% of processes within one standard deviation and 96% within two standard deviations from the mean. The distribution is a little broader than before, however the standard deviation is smaller. Furthermore, the percentage of outliers has improved and is even less than with the SFC method. 13 or 1.5% of processes have more connections than three standard deviations from the mean. The highest edge count is 50.

To complete the evaluation of the process connectivity, the communication links are again illustrated between the respective MPI ranks. Figure 5.28 shows the connectivity matrix at time $t = 38$ s for the four different node counts. The connectivity in general is mostly clustered around the main diagonal, which is beneficial. Although not guaranteed, numerically close MPI ranks usually are run on physically close components. For example on cores on the same microprocessor. This entails faster communication. This beneficial clustering around the main diagonal with less off-diagonal entries is even more pronounced for higher nodes counts.



(a) 4 nodes (b) 8 nodes

(c) 16 nodes (d) 32 nodes

Figure 5.28: Process connectivity at time $t = 3.8$ s.

Finally, the grid connectivity results for this test case are presented in the following. In Figure 5.29 the inter-process edges for the SFC and HD methods and the different node counts are shown. Again, all values are taken at time $t = 38$ s. The HD method exhibits 20,183 inter-process edges for 4 nodes, 23,965 for 8 nodes, 29,499 for 16 nodes and 35,872 for 32 nodes. The SFC method exhibits 14,530 edges for 4 nodes, 19,114 for 8 nodes, 25,503 for 16 nodes and 33,182 for 32 nodes. The amount of inter-process grid edges is lower for all measured node counts when using the SFC strategy. However, the percentage increase is higher. HD's edge count increases between 19 and 23 percent for every doubling of the nodes. SFC's edge count increases between 30 and 33 percent. If this behaviour persists, SFC's total edge count will overtake HD's count when the number of nodes is further increased.

The distribution of spatial and hierarchical edges stays relatively constant for HD. Inter-process hierarchical edges fluctuate between 18 and 22 percent of all inter-process edges. The corresponding spatial edges fluctuate between 78 and 82 percent respectively. The percentage of hierarchical edges is lower for the SFC repartitioning. However, it increases with more nodes used from 8 to 17 percent. Correspondingly, the proportion of inter-process spatial grid connections decreases from 92 to 83 percent. If this behavior continued, the percentages between the two repartitioning strategies would even out on the next doubling of nodes. Furthermore, a subsequent increase of nodes could lead to a higher percentage of hierarchical edges of the SFC strategy compared to the percentage observed when using the HD strategy.

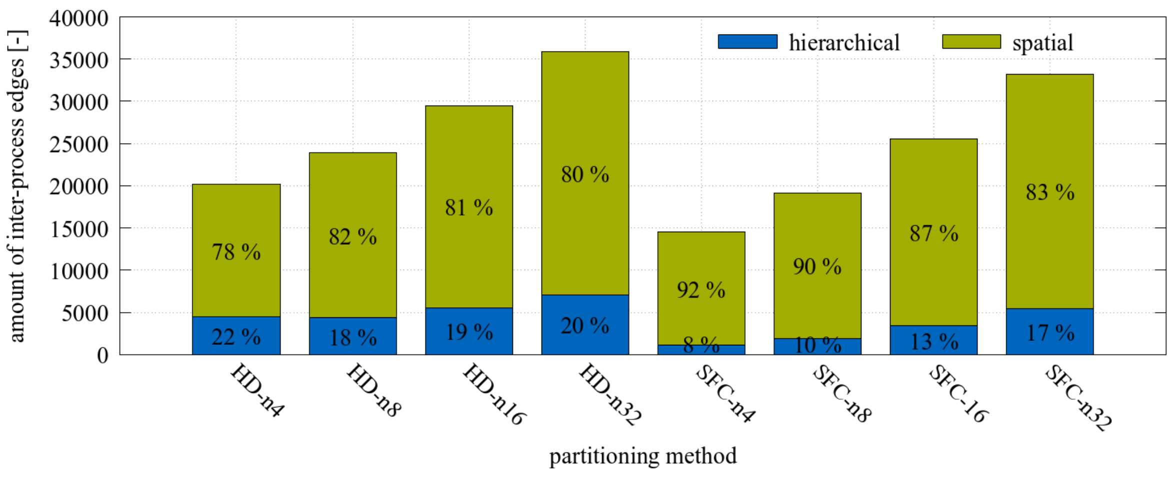


Figure 5.29: Amount of total inter-process edges between grids, divided into spatial and hierarchical edges for both repartitioning strategies and different node counts at time $t = 3.8$ s.

5.2.4 Migration

To conclude the observations for this test example, the migration statistics are presented. Figure 5.30 illustrates the total amount of grid transfers from one process to another for each timestep. The statistics at the end of the runtime are given in Table 5.2.

Again, the SFC method, incurs massive redistribution costs that linearly scale with the amount of nodes used. In total, 2.5 million grid migrations are necessary to reach the computed partitions using 4 nodes. At the top end, with 32 nodes, more than 16.3 million migrations are necessary. The amount of individual grids transferred during their lifetime rises from 93 to 99 percent. Of these grids, each one is migrated 30 times on average on 4 nodes. The most migrated grid was transferred 763 times. On 32 nodes, the average migration per grid is 183 and the single most migrated grid was transferred 2,072 times. There are 3,942 repartitioning steps during the runtime. At the top end, a grid has been transferred at every second repartitioning. Because of the massive migration costs, which linearly scale with the amount of processes, an SFC-based repartitioning is not feasible in practice for large clusters.

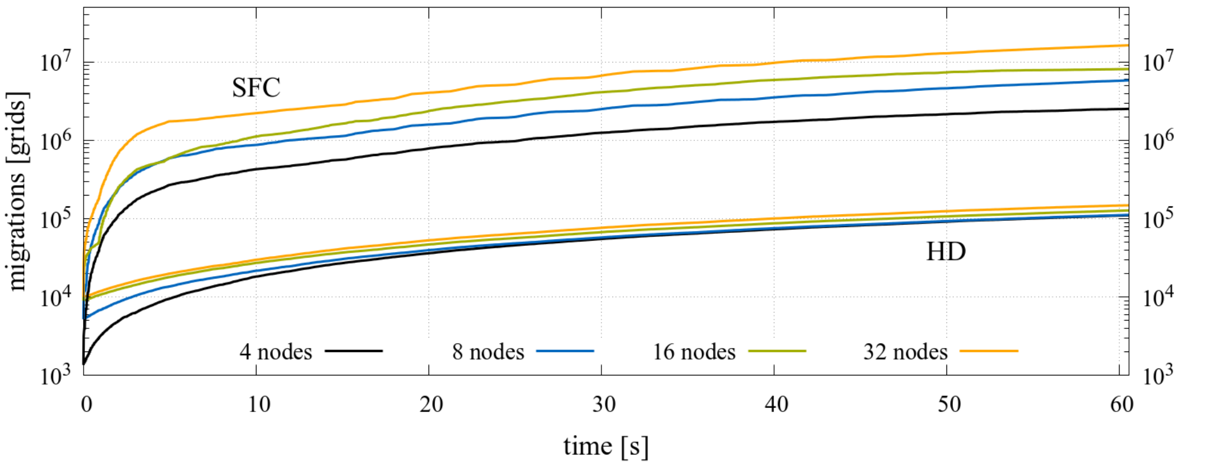


Figure 5.30: Total amount of migrations per timestep for SFC and HD strategies for different amounts of nodes.

The diffusion scheme incurs only a fraction of the repartitioning costs. On 4 nodes, the scheme accumulates 111 thousand migrations in total, only 4.3% of the migrations of SFC. The amount of migrations is also affected by an increase of nodes. On 32 nodes, the total amount of migrations rises to 149 thousand. Per repartitioning, only 28 grids are migrated when using 4 nodes. The highest amount of migrations per repartitioning step is 118. The average rises slightly to 35 migrations on 32 nodes. The highest migration count per repartitioning step is almost unchanged at 120 migrations. Therefore, the increase in total migrations can be attributed to the rise in individual grids migrated. Even though each grid is migrated less on average, 4.44 times on 4 nodes compared to 2.18 on 32 nodes, the number of individual grids transferred rises from 25 thousand (27.6%) to 68 thousand (75.5%). In the observed range, the redistribution costs of the diffusion scheme are therefore not completely independent from the domain size. However, compared to a global method, the costs are massively lower, with much leeway until the costs become an issue. Furthermore, because the rising redistribution costs can be attributed to the increase in individual grids migrated, there is an upper bound to this increase, when all grids participate in the redistribution.

The overall runtimes give a hint of the impact of the added migration costs. Workload balance is optimal for the SFC distribution and the overall number of edges is between 35% and 45% lower. Therefore, one can expect a lower iteration time, less communi-

Table 5.2: Migration statistics for the AMB test example.

SFC	4 nodes	8 nodes	16 nodes	32 nodes
total migrations	2,554,321	5,843,808	8,144,439	16,373,183
individual grids migrated	83,853	86,723	88,327	89,478
ratio [%]	92.58	95.88	97.65	98.93
average migrations per step	648	1,482	2,066	4,154
max migrations per step	3,464	8,488	11,411	17,380
average migrations per grid	30.46	67.38	92.21	182.99
max migrations per grid	763	860	1,284	2,072

HD	4 nodes	8 nodes	16 nodes	32 nodes
total migrations	110,826	111,968	127,629	148,632
individual grids migrated	24,913	38,238	56,357	68,245
ratio [%]	27.55	42.28	62.23	75.46
average migrations per step	28	28	32	35
max migrations per step	118	106	114	120
average migrations per grid	4.44	2.93	2.26	2.18
max migrations per grid	72	72	56	54

cation and consequently lower overall runtimes without redistribution costs when using the SFC repartitioning strategy. The actual runtimes with redistribution costs are given in table 5.3. One observes that not only the time save given by the better balance and favourable connectivity has been completely evened out, but the diffusion approach is considerably faster for all measured cases. The increase in migration volume for higher node counts also has compounding effects on the runtimes. The total runtime grows from 50 min using 4 nodes to 1:43 min using 32 node. The percentage increase in runtime is 14% at 4 nodes, increasing to a runtime increase of 92% at 32 nodes.

Table 5.3: Average runtimes in [h:min:sec] for the additive manufacturing benchmark.

	4 nodes	8 nodes	16 nodes	32 nodes
SFC	12 : 30 : 53	7 : 31 : 55	4 : 43 : 16	3 : 35 : 15
HD	11 : 40 : 32	6 : 15 : 28	3 : 24 : 41	1 : 52 : 13

5.2.5 Short Summary

The distinct feature of this test example is a very small localised phenomena that travels only through parts of the domain. While the laser track draws the initial square, the distances the laser covers are larger. Afterwards, during the hatch the distances become smaller. To allow for an evaluation of the influence of the number of participating processes on the repartitioning, this testcase has been deployed to 4, 8, 16 and 32 nodes with 112, 228, 448 and 896 processes respectively.

Looking at workload balance, the key aspect here is that the diffusion repartitioning still performs reasonably well and even limits outliers for very small average grid counts. However, especially for 32 nodes, the granularity for a good workload balance is too coarse. 21 grids on average per process do not give enough leeway to reach a balanced distribution. Even relatively small outliers considering the total overall load have a high impact on the distribution, as they impose between half and one and a half times the average load. Not only is a coarse granularity detrimental to the workload balance, it also has an impact on the solution procedure. Frisch argues that the best results can be achieved upwards of 100 grids per process [113], which is only the case using 4 nodes for the given domain decomposition.

When it comes to process connectivity, the diffusion scheme benefits from an increase in node counts, whereas the SFC method sees either no improvement or produces slightly worse results. Nevertheless, all measurements so far still show an advantage for the SFC scheme. The total amount of edges is influenced by the locality of the phenomena for the diffusion scheme. As expected, the SFC, as a global method is not affected. The advantage of the SFC method decreases from 1.7 times as many connections to 1.2 when using 32 nodes. The standard deviation improves when using more processes in the diffusion approach, both in value and volatility. The distribution quality improves too, producing less outliers. Even compared to the SFC distribution, which produces more outliers. Concerning the overall inter-process grid connectivity, even though the SFC shows better results for all measured node counts, it appears less scalable and the HD method will outperform it for higher node counts.

The redistribution costs incurred by the SFC method scale linearly with the number of processes used and are massive in comparison to the diffusion approach. While the number of migrations still increases with more processes used for the diffusion, the increase is marginal. Even though the workload balance and connectivity measures favour the SFC partitioning, the added migration volume leads to longer runtimes compared to the diffusion strategy, which is almost twice as fast at 32 nodes.

5.3 Test Example - Von Kármán Vortex Street

The third and last example is one of the benchmark cases proposed by Schäfer and Turek within the DFG priority research program ‘Flow Simulation on High-Performance Computers’ [273]. Here, the incompressible Navier–Stokes equations are solved to simulate laminar flow around a cylinder. For high enough Reynolds numbers vortexes shed behind the obstacle and result in a pattern known as von Kármán vortex street. To accurately capture the flow structures, a refinement is performed around regions of high vorticity.

5.3.1 Test Setup

Here, the mathematical model is represented by the Navier–Stokes equations for incompressible Newtonian flows. These consist of the continuity equation and the momentum equations for three dimensions in differential form:

$$\nabla \cdot \vec{u} = 0 \tag{5.2}$$

$$\frac{\partial \rho_\infty u_i}{\partial t} + \nabla \cdot (\rho_\infty u_i \vec{u}) = \nabla \cdot (\mu \nabla u_i) - \nabla \cdot (p\vec{e_i}) + b_i. \tag{5.3}$$

$\vec{u}$ describes the velocity vector of the flow field, t represents the time, ρ_∞ the density of the fluid (assumed constant over the entire domain), u_i is the velocity in direction i, μ the dynamic viscosity, p the pressure, b_i are some interior body forces in direction i, and $\vec{e_i}$ is the unit vector in direction i. To solve these equations a fractional step method proposed by Chorin [58] is used, which allows to decouple the velocity and pressure fields. Ignoring the pressure gradient, an intermediate velocity field that does not satisfy the incompressibility constraint is computed. Using the continuity equation 5.2 allows to formulate a Poisson's equation for the pressure. The pressure is then used to correct the intermediate velocity field to satisfy the incompressibility constraint.

The pressure Poisson equation is the computationally most intensive part. Here, the custom multigrid method detailed in section 3.6.2 is used. For spatial discretisation a finite volume method that locally degenerates into finite differences due to the block substructuring of the domain into regular Cartesian grids is applied. Time stepping is done by a two-step Adams–Bashforth method that is accurate to second-order.

The benchmark setup is a three-dimensional channel flow with a cylinder obstacle near the inlet on the left-hand side channel boundary. Figure 5.31 illustrates the configuration for the benchmark. The channel has a length of $l = 2.2$ m and a height and width of $h = w = 0.41$ m. The cylinder's diameter is $d_{cylinder} = 0.1$ m with its center located at a miniscule deviation from half width at $y_{cylinder} = 0.2$ m and $x_{obstacle} = 0.5$ m.

Schäfer and Turek describe three different benchmark scenarios each for two and three dimensions. 2D-1 and 3D-1 with $Re = 20$ produce a steady solution, 2D-2 and 3D-2 with $Re = 100$ produce an unsteady solution and 2D-3 and 3D-3 with $Re = 100$ and transient inflow conditions. However, they themselves argue for higher Reynolds numbers in three dimensions for the unsteady solutions, as the flow tends to become almost stationary. Therefore, for the present example a Reynolds number of $Re = 150$ was chosen.

The Reynolds number is defined by

$$Re = \frac{\overline{u}D}{\nu}. \tag{5.4}$$

$\overline{u}$ is the characteristic velocity $\overline{u} = \frac{4}{9}u_{max}$, D is the characteristic length scale, here $D = d_{cylinder}$ and ν is the kinematic viscosity of the fluid. The kinematic viscosity for water is $\nu = 10^{-3}$ m^2/s. To get $Re = 150$, the maximum velocity in x-direction needs to be $u_{max} = 3.375$ m/s.

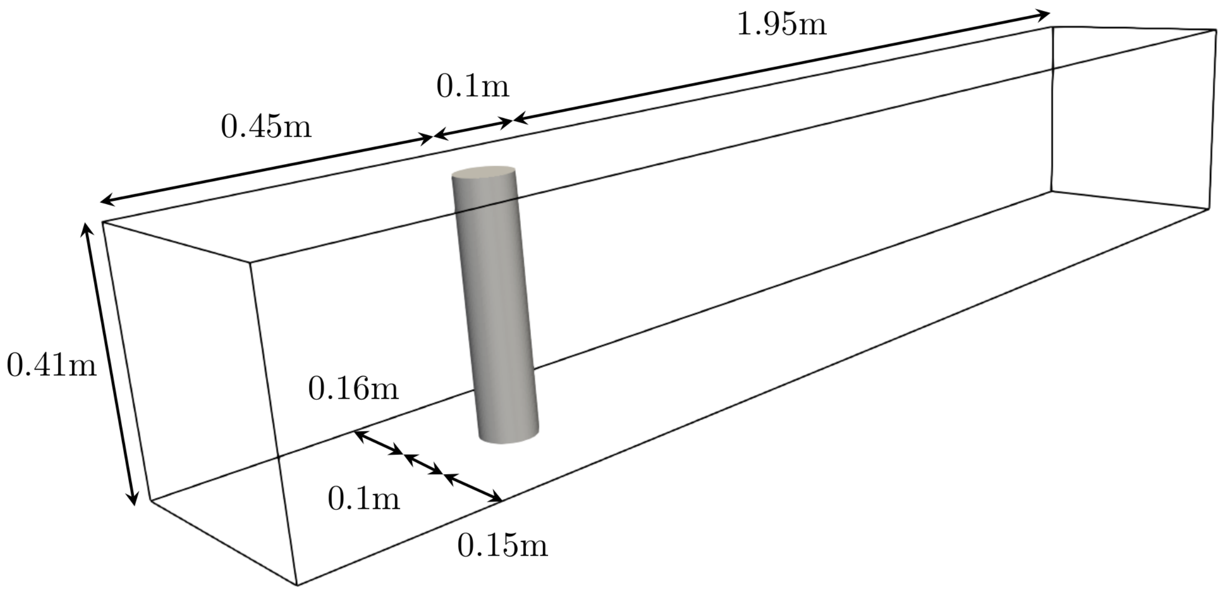


Figure 5.31: Configuration for flow around a cylinder test example.

The inlet velocities in x-, y- and z-direction (u_x,u_y and u_z) are given by

$$u_x(x,y,z) = 16 \cdot u_m \cdot yz \frac{(w-y)(h-z)}{w^2 \cdot h^2} \text{ m/s}, \quad u_y = 0, \quad u_z = 0. \tag{5.5}$$

The initial conditions in the flow regime are given by the same formula, resulting in a parabolic flow profile. Top, bottom, north and south walls are prescribed with a no-slip boundary condition, the east end of the domain serves as an outlet.

The channel is decomposed using the space-tree approach with a subdivision spacing for the root node of $(4, 1, 1)$ and a spacing of $(2, 2, 2)$ for all other nodes. Refining the root node such that the most elongated direction is subdivided into four smaller nodes, serves to generate more cubically shaped grids. The domain is uniformly refined to depth $d = 3$, which results in 256 total nodes and 288 leaf nodes on refinement level $d = 3$. Each node is discretised with a grid, which consists of $8 \times 8 \times 8$ cells.

As the phenomena of interest is the vortex shedding behind the cylinder obstacle, the vorticity is used as an indicator for refinement or coarsening. It is calculated as the curl of the flow velocity $\vec{u}$:

$$\omega = \nabla \times \vec{u} = \begin{pmatrix} \frac{\partial u_x}{\partial y} - \frac{\partial u_y}{\partial z} & \frac{\partial u_x}{\partial z} - \frac{\partial u_z}{\partial x} & \frac{\partial u_y}{\partial x} - \frac{\partial u_x}{\partial y} \end{pmatrix}. \tag{5.6}$$

If the maximum vorticity magnitude in a grid exceeds a value of 70 m/s, the node is set for refinement up to a maximum depth of $d = 5$. Conversely, a coarsening is ordered if the value goes below 50 m/s until the initial uniform refinement depth $d = 3$ is reached. As always, 2:1 domain balance is required. Before running the simulation, two refinement steps are carried out using the initial flow profile. This results in a refinement around the obstacle. All generated grids are distributed among 112 processors or 4 nodes of the

CoolMUC-2 cluster segment. Following the domain generation method outlined in 3.4, the initial partitioning follows the Z-order SFC distribution.

The simulation ran for a total of $t_{end} = 5.5$ s. The timestep size is determined by the Courant–Friedrichs–Lewy (CFL) condition. Usually, one would use the CFL condition to adaptively increase or decrease the timestep size, while ensuring convergence of the solver. In this case, the aim was to keep the timestep size constant. Therefore, the CFL condition was used once, to determine the allowed timestep size at the start of the simulation, after the initial refinement and with the initially prescribed velocities. To ensure convergence, the used timestep size was taken as ten percent the allowed size, which turns out to be $dt = 9.7 \cdot 10^{-5}$ s. A domain update is performed every $5 \cdot 10^{-3}$ s, or around every 50 timesteps. In contrast to the previous test cases, the domain repartitioning, which uses a single iteration of the diffusion method is run every timestep. This serves no practical use, it however allows a closer consideration of the progression of the diffusion. The SFC repartitioning, used as a comparison, is run after every refinement and coarsening step as before.

Figure 5.20 shows the evolution of the number of grids during the runtime of the test. At time $t = 0$ s, after the two initial refinement steps, the domain has 732 leaf grids and 837 grids in total. This amounts to roughly $375,000$ cells on the finest level with 1.5 million degrees of freedom and $429,000$ cells with 1.7 million degrees of freedom in total. The vortex street develops approximately until time $t = 1$ s. The total number of grids is steadily increasing up to approximately 5,000 during this time. After the vortex street has developed, refinement and coarsening follow each vortex through the domain and balance each other out. The total amount of grids stays roughly constant between 5,000 and 5,500. In terms of cells, this amounts to between 2.56 and 2.82 million cells with between 10.24 and 11.26 million degrees of freedom respectively. The variability after the vortex street has developed, is around 11%. The highest computational intensity, i.e. the highest number of grids is found at time $t = 5.39$ s.

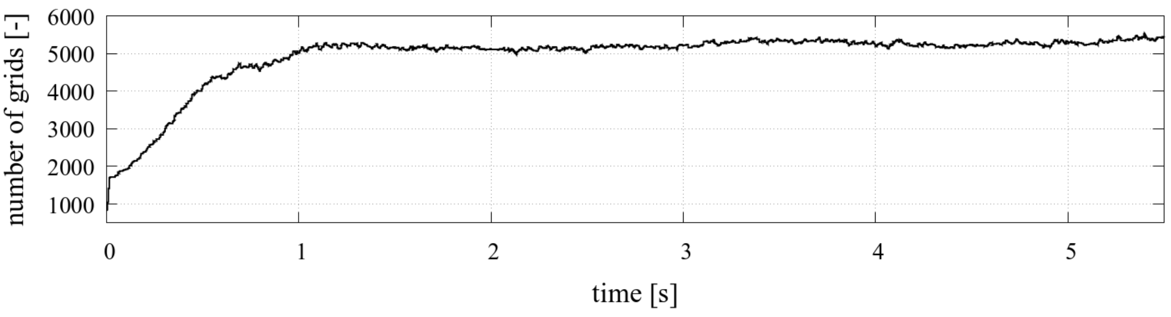


Figure 5.32: Number of grids in the domain per timestep.

Figures 5.33 and 5.34 aim to give an overview into the development and evolution of the vorticity structure within the channel. In Figure 5.33, an isometric view into the domain is depicted. Only the cells with vorticity magnitude higher than 50 m/s are shown. The wireframe illustrates the structure of the grids on the deepest refinement level $d = 5$, which follow these regions of high vorticity magnitude. The plots in Figure 5.34 show slices perpendicular to the z-axis at half height and similarly aim to illustrate the evolution of the vorticity. All cells are coloured by their vorticity parallel to the z-axis.

In the following, the diffusion-based repartitioning with highest degree target determination (HD) is examined in terms of workload balance, connectivity and migration frequency for this testcase. Again, the Z-order SFC repartitioning method serves as a baseline.

5.3.2 Workload Balance

The first measurement again shows the course of the workload balance in terms of standard deviation σ during the runtime of the simulation. This is illustrated in Figure 5.35. The curve for the SFC partitioning is purposely omitted. The global method achieves a perfect workload balance with discrepancies of at most one grid per process. Therefore, σ_{SFC} is less than one. When using the diffusion strategy, the balance in terms of σ rises and shows a high volatility between values of 4 and 8 during the initial consolidation of the domain as the vortex street develops. As soon as the rate of change in the domain slows down and the vortex street has developed at around $t = 1$ s, the workload balance steadily decreases down to roughly $\sigma = 2.5$ at time $t = 2.1$ s. From here, the standard deviation stays in the range between 2.5 and 3.5 with a slight decrease towards $\sigma = 2$ towards the end of the runtime.

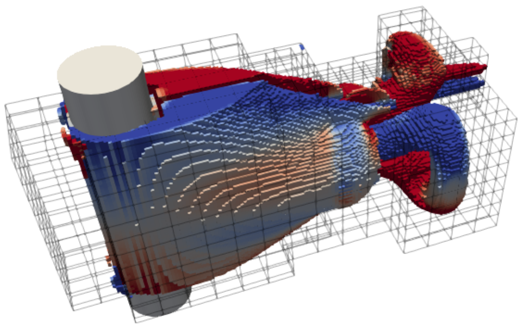
In terms of relative standard deviation, illustrated in Figure 5.36, a value of 35% is measured at the beginning, which then steadily decreases down to 5% at time $t = 2$ s.

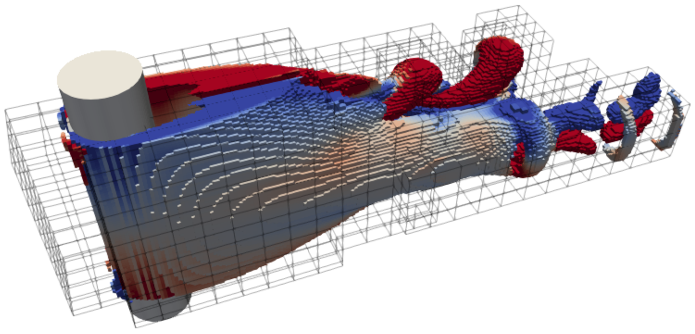
As mentioned above, coarsening and refinement is evaluated approximately once every 50 timesteps, a diffusion rebalancing however takes place every timestep. This reveals the progression towards a local optimum. In Figure 5.37, a zoomed-in view of the progression of the standard deviation between $t = 0.5$ s and $t = 0.55$ s is shown. The flattening curves make clear that the first repartitioning step returns the most reduction and further steps have diminishing returns. After most AMR cycles, it takes around 4 steps to reach a local optimum. The maximum amount of steps until the distribution stays constant during the complete simulation time are 7.

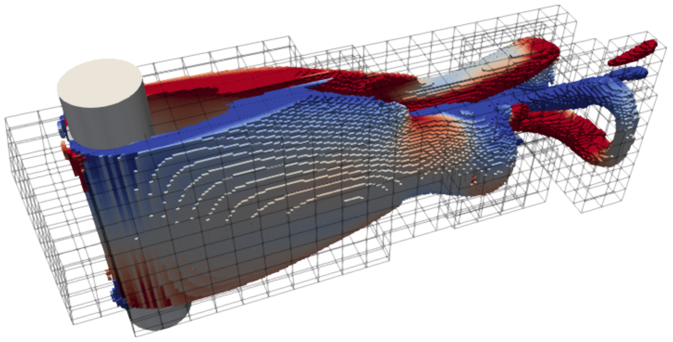
To complete the evaluation of the workload balance, a closer look at the distribution at time $t = 5.39$ s, when the highest amount of grids exist, is provided in Figure 5.38. The average amount of grids μ at this timestep is 48.5, the standard deviation is $\sigma = 2.5$. The distribution is comparably good. The three highest samples are 47, 48 and 49 grids per processes with a frequency of 23, 26 and 22. This means 71 out of 112 processes have an amount of grids very close to the mean. In total, 80% of all processes lie within one standard deviation from the mean and 94% are within two. The process with the smallest amount has 40 grids. The maximum amount of grids per process in the domain is 58, which two processes have.

5.3.3 Connectivity

The examination again commences by analysing the connectivity between processes. In Figure 5.39 all relevant measurements for both the SFC and HD repartitioning strategies


(a) $t = 0.4$ s


(b) $t = 0.8$ s


(c) $t = 2.9$ s

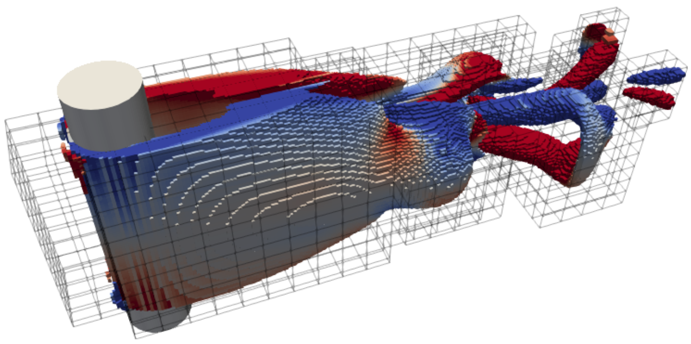
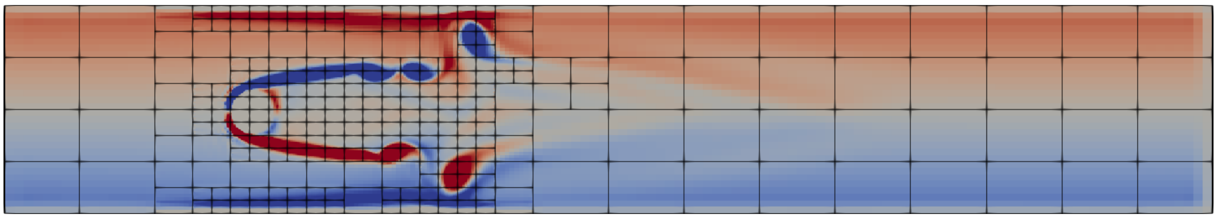
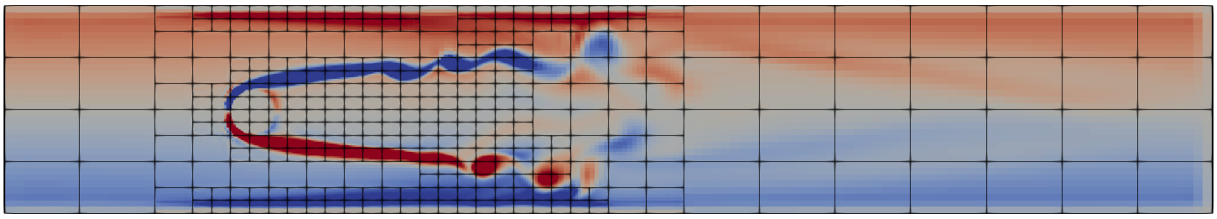
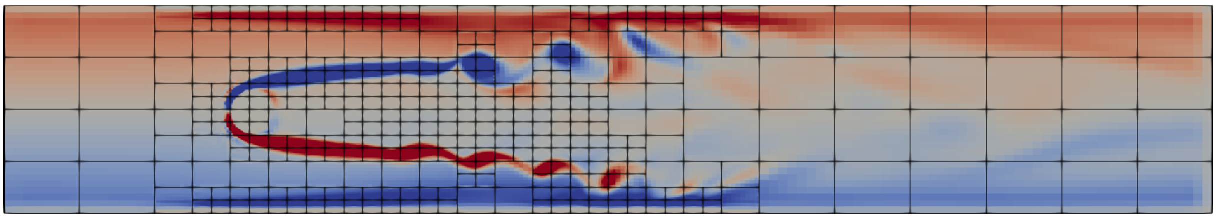
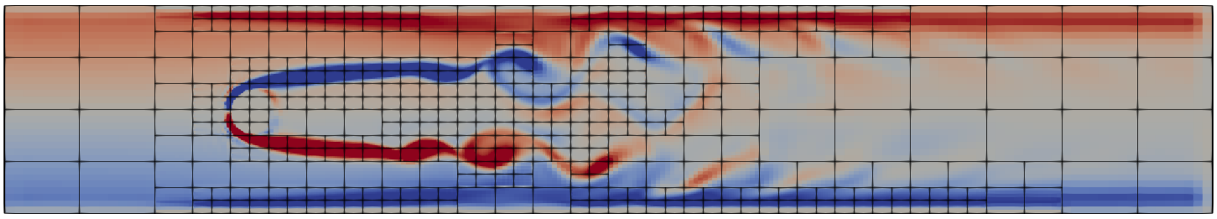

(d) $t = 5.3$ s

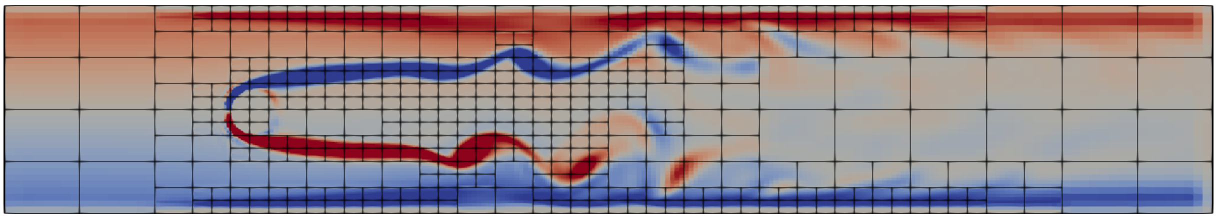
Figure 5.33: Illustration of the vorticity at selected timesteps. Cells with vorticity magnitude higher than 50 m/s are shown. Red colouring marks a positive vorticity in x-direction, blue marks a negative vorticity in x-direction. The plots are overlaid with the grid structure on the deepest refinement level $d = 5$.


(a) $t = 0.4$ s


(b) $t = 0.6$ s


(c) $t = 0.9$ s


(d) $t = 1.8$ s


(e) $t = 3.0$ s

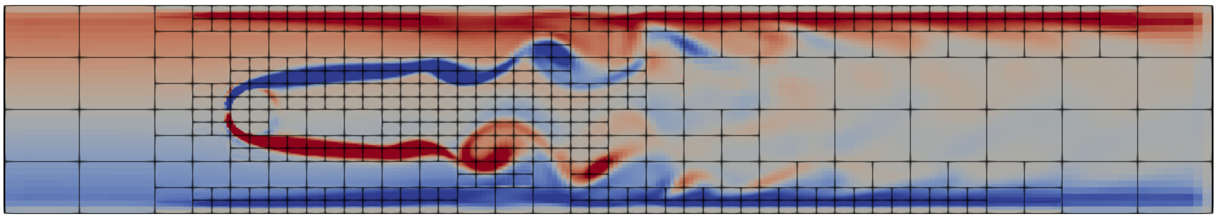

(f) $t = 4.9$ s

Figure 5.34: Slice through the domain perpendicular to the z-axis at half height at selected timesteps. Cells are coloured according to their vorticity parallel to the z-axis. Red colouring marks a positive vorticity, blue marks a negative vorticity. The plots are overlaid with the grid structure.

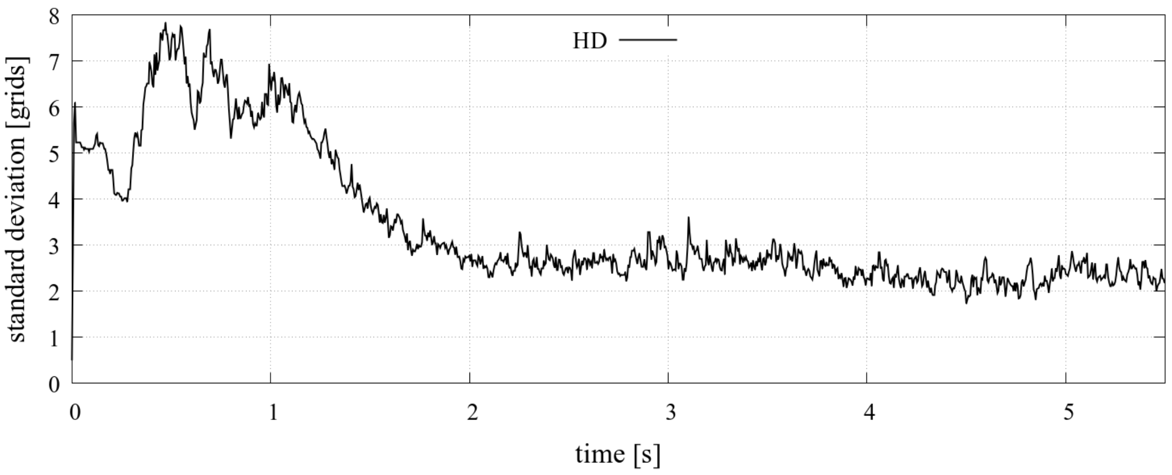


Figure 5.35: Standard deviation of grids per process measured at every timestep using HD diffusion repartitioning.

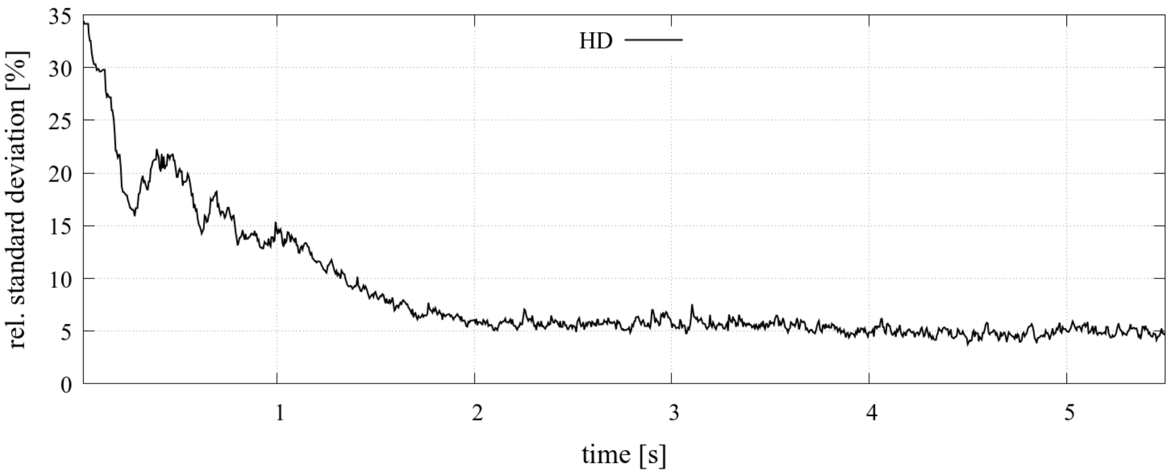


Figure 5.36: Relative standard deviation of grids per process measured at every timestep using HD diffusion repartitioning.

are shown. The total number of connections (edges) between processes (5.39a), the standard deviation of edges per process (5.39b) and the relative standard deviation (5.39c).

Similar to the workload balance, for the HD strategy, the initial number of connections rises in the beginning of the simulation during the formation of the vortex street from roughly 500 to 1,050 edges. However only up to time $t = 0.6$ s. Afterwards, the total number of edges stays relatively constant, with a small decrease to around 1,000 edges until the end of the simulation. The SFC also shows a small increase up to time $t = 0.35$ from roughly 500 to 580 edges and stays at this level for rest of the runtime. The rise in standard deviation is equivalent to the rise in total number of edges. σ_{HD} rises from a value of 3 to 8 and stays in the range between 8 and 9 roughly until time $t = 3.8$ s and then drops slightly to a range between 7 and 8 for the remainder of the runtime. σ_{SFC} shows a slight increase in the beginning at time $t = 0.35$ s from 3 to 4, then slightly decreases to

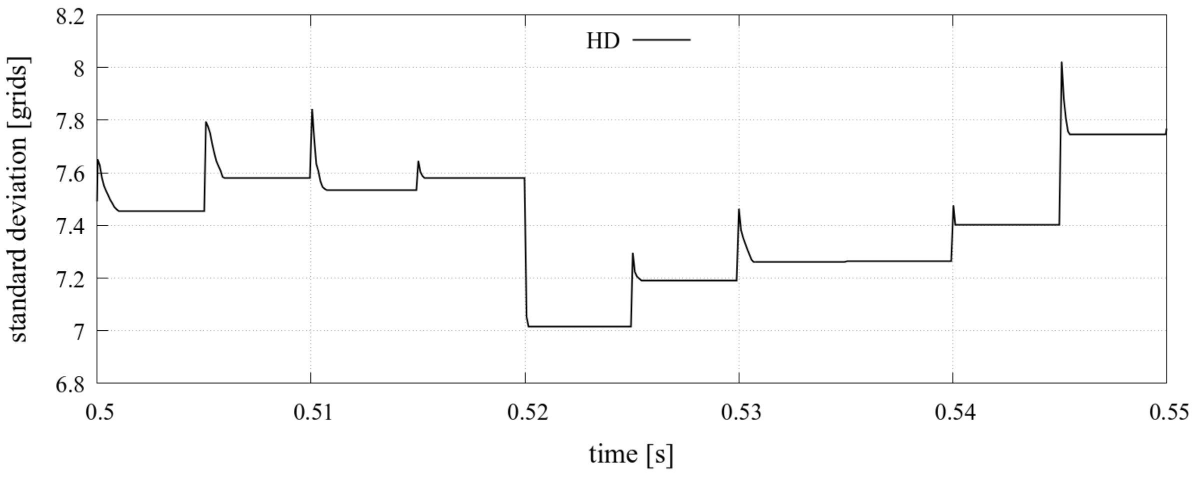


Figure 5.37: Zoomed in view of the standard deviation of grids per process measured between time $t = 0.5$ s and $t = 0.55$ s using HD diffusion repartitioning.

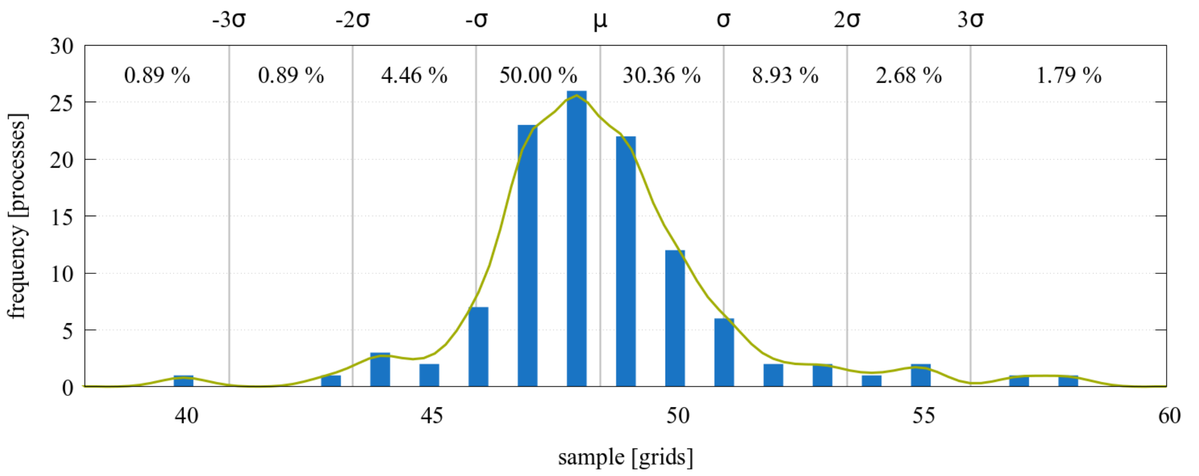
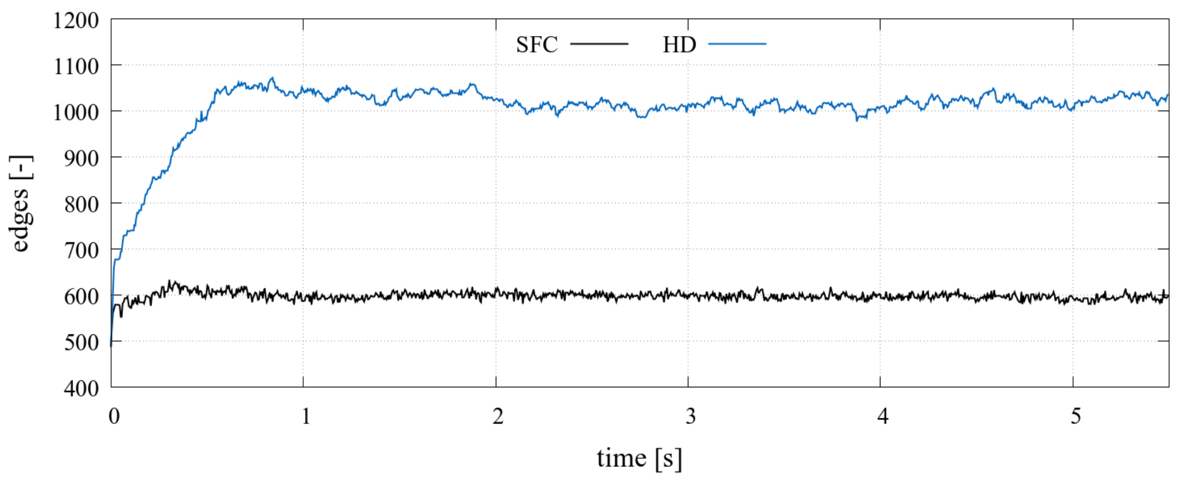


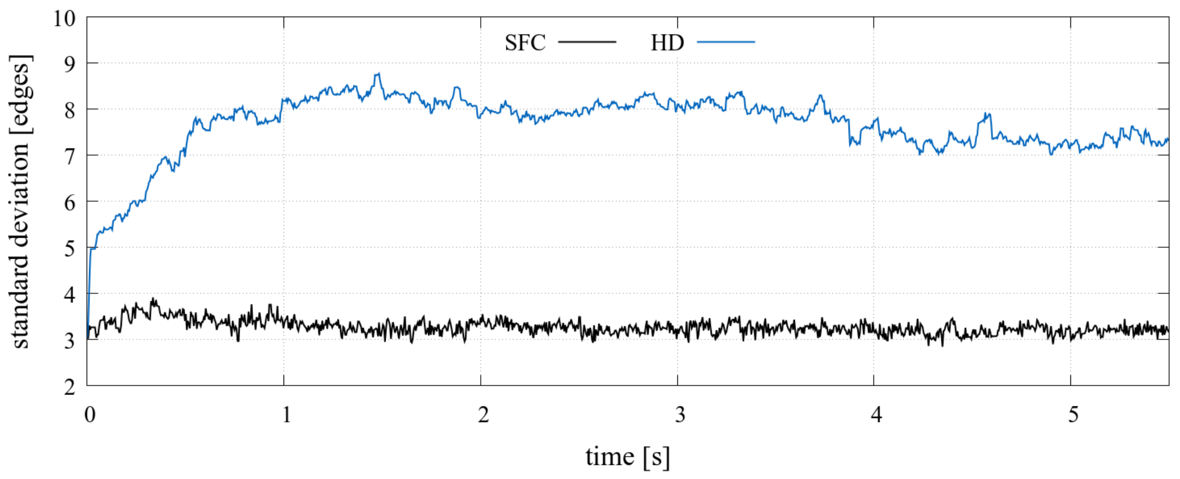
Figure 5.38: Histogram of the frequency distribution of grids per process at time $t = 5.39$ s with HD diffusion repartitioning. The histogram is overlaid with the smoothed kernel density estimate using a Gaussian kernel.

a relatively constant trend between values of 2 and 2.5. The relative standard deviation, as a combination of the two former measurements reveals a behaviour seen also in the two former testcases. Both strategies show roughly constant relative standard deviations in the range between 40 and 45 percent for HD and between 30 and 35 percent for SFC. This suggests that regardless of the testcase, the SFC and HD strategies converge towards a fixed quality of the process connectivity.

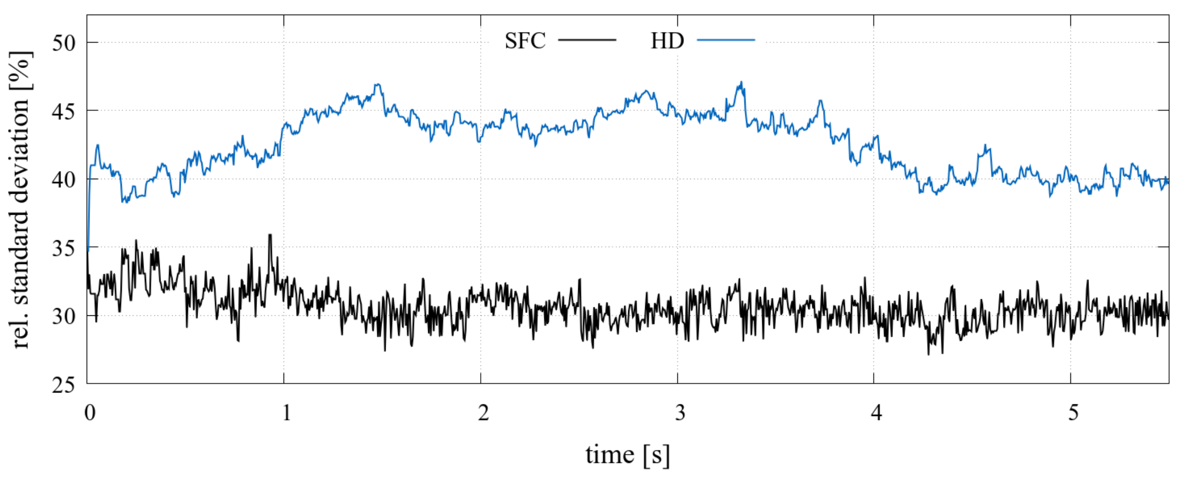
A closer look into the distribution of process edges at time $t = 5.39$ s is given in Figure 5.40. The quality of the SFC distribution is quite good with a rather narrow distribution around the average amount of edges per process μ_{SFC} of 10.6. With $\sigma_{SFC} = 3.2$, 75% of all processes lie within one standard deviation from the mean and 93% are within two. Still, there is a considerable amount of processes (6%) in the range between two and three



(a) Total number of edges.



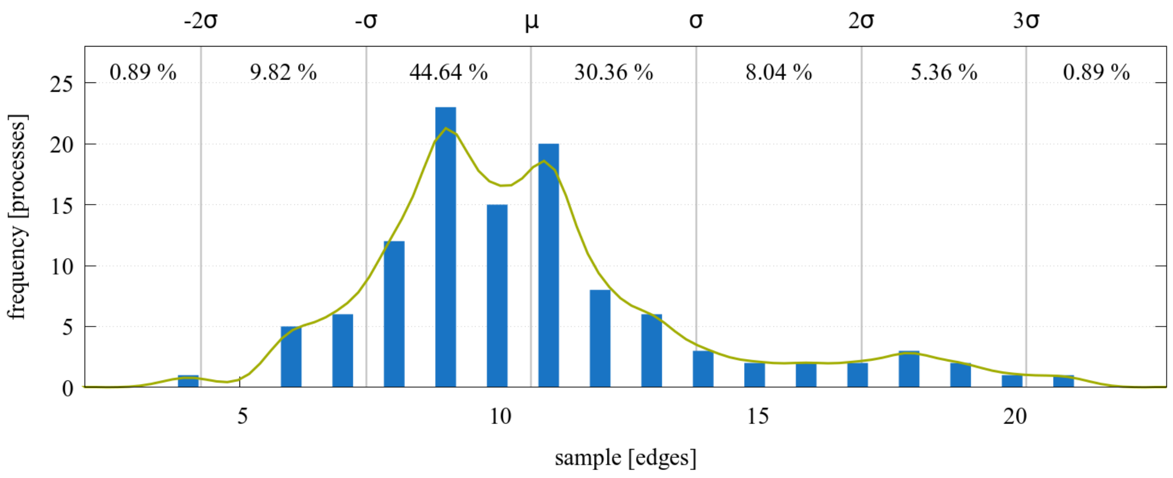
(b) Standard deviation.



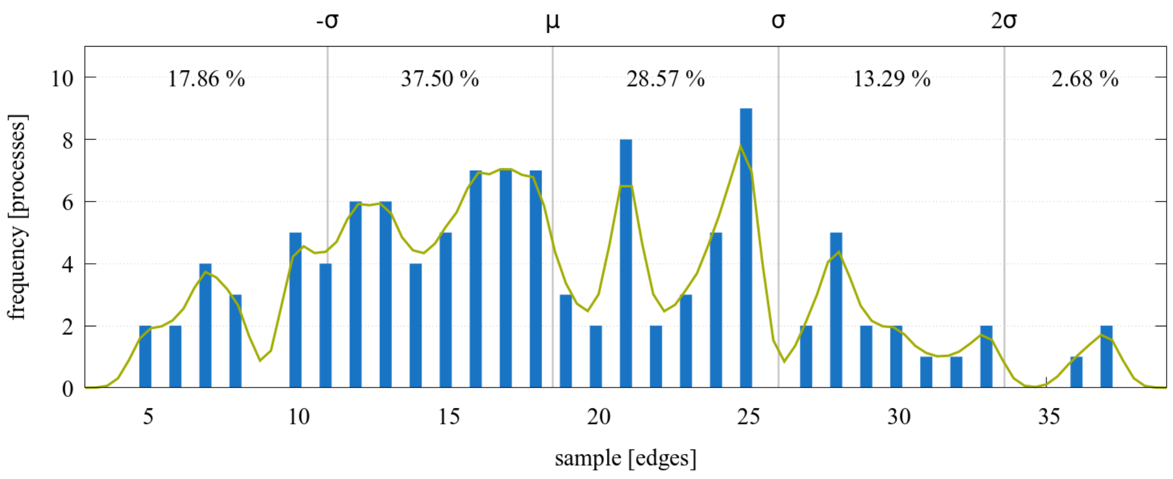
(c) Relative standard deviation.

Figure 5.39: Process connectivity in the domain per timestep using the HD and SFC repartitioning strategies.

standard deviations and a single outlier even farther from the mean. This outlier has 21 neighbourhood connections. In comparison, the HD distribution has entirely different characteristics. The distribution is more narrow due to a higher standard deviation σ_{HD} of 7.5 and mean μ_{HD} of 18.6. It is however also more shallow. Within one standard deviation from the mean are 66%, within two are 97% of all processes. Three processes lie in the range greater than two standard deviations. One of them has 36 edges, the other two have 37.



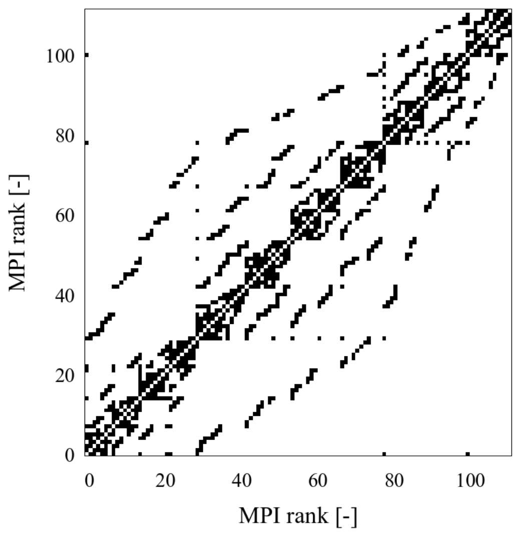
(a) SFC



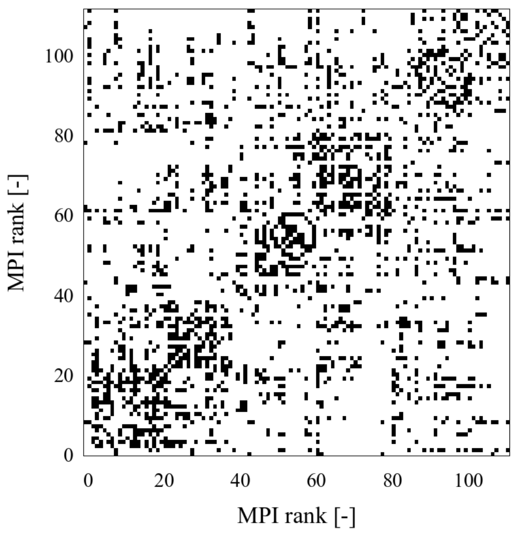
(b) HD

Figure 5.40: Histograms of the frequency distribution of edges per process at time $t = 5.39$ s with SFC and HD repartitioning strategies. The histogram is overlaid with the smoothed kernel density estimate using a Gaussian kernel.

To conclude the observation into neighbourhood relations between processes, the connectivity between the individual MPI ranks is illustrated in Figure 5.41 at time $t = 5.39$ s. The plot for the SFC repartitioning shows the characteristic clustering around the main diagonal. Compared to previous examples, the clustering is not as narrow, which suggests that even the SFC method cannot reach an optimal clustering here. These difficulties become more clear looking at the connectivity matrix for the HD method. The clustering from the initial partitioning has been mostly deteriorated with only a faint residue around the main diagonal.



(a) SFC



(b) HD

Figure 5.41: Process connectivity at time $t = 5.39$ s.

The measurements of the grid connectivity are rather unremarkable and follow the observations made for the previous testcases closely. Figure 5.42 illustrates the amount of inter-process edges between grids at time $t = 5.39$ s. In total, the SFC method exhibits 5,714 edges and the HD method has 8,043 edges. The same increase as seen previously of 40%. Also the distribution between spatial and hierarchical grid edges is unsurprising, with a ratio of 83 to 17 percent for the SFC strategy and 72 to 28 percent for the HD strategy.

This concludes the observations into process and grid connectivity. The main conclusions here are that the different partitioning strategies are somewhat independent from the problem and tend towards specific characteristic values. That includes the relative standard deviation of connections between processes, the relative number of inter-process edges between SFC and HD partitioning methods as well as their respective ratios between hierarchical and spatial edges.

5.3.4 Migration

Finally, the migration statistics are presented. Figure 5.43 illustrates the total amount of grid transfers from one process to another for each timestep. The statistics at the end of the runtime are given in Table 5.4.

Again, no new findings can be gathered from these measurements. Even though the HD redistribution is run roughly four times as frequent – it runs after every timestep but only redistributes four times per AMR cycle on average – the difference in the amount of

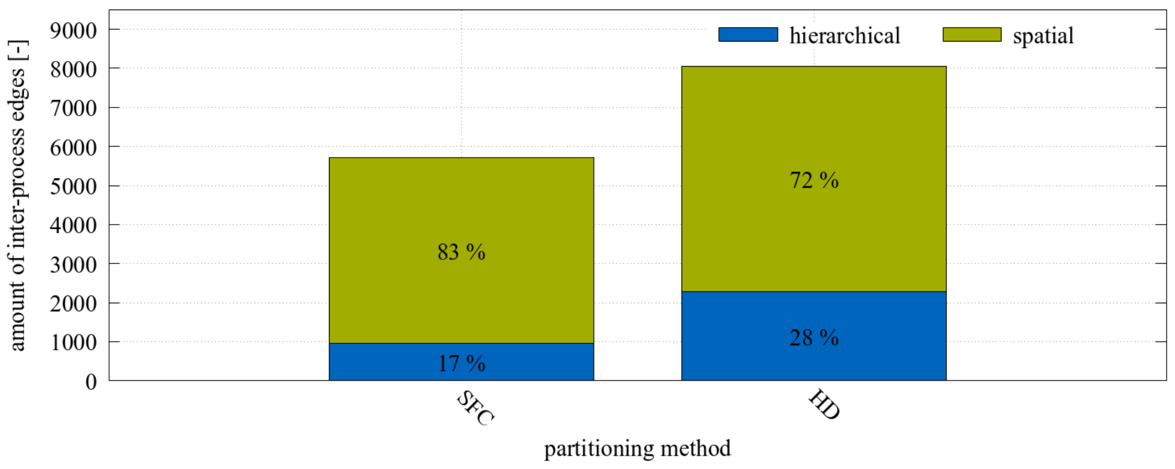


Figure 5.42: Amount of total inter-process edges between grids, divided into spatial and hierarchical edges for both repartitioning strategies and different node counts at time $t = 5.39$ s.

migrations is an order of magnitude. SFC accumulates 900 thousand grid transfers, 30 times as many as the HD method with 30 thousand in total. Using the SFC strategy, close to 90% of all grids get migrated at least once during their lifetime. On average each grid moves from one process to another 37 times. To reach the partitions computed by the diffusion method, only 21% of all grids get transferred ever, on average 4.3 times.

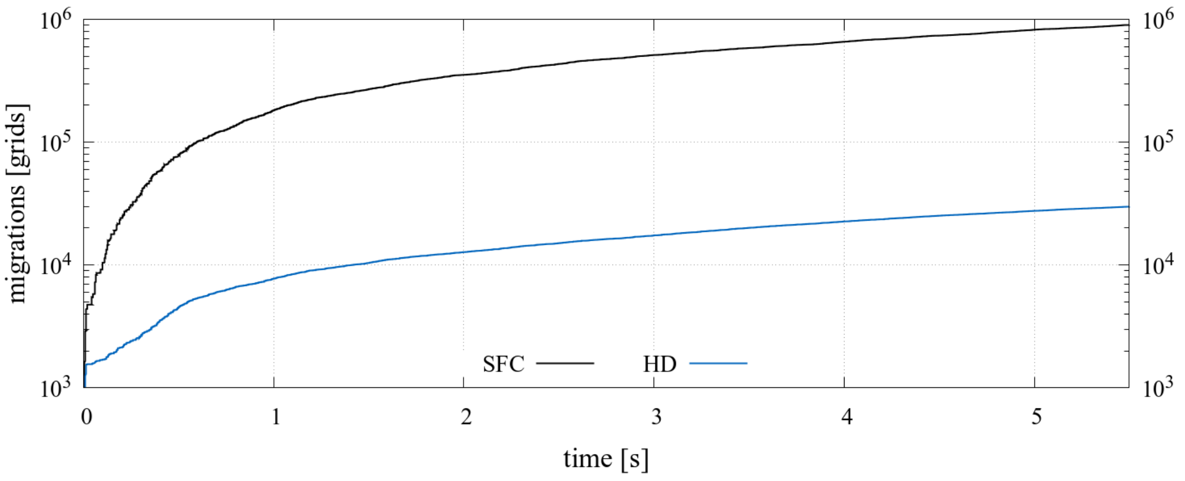


Figure 5.43: Total amount of migrations per timestep for SFC and HD strategies.

5.3.5 Short Summary

This example solves the most complex mathematical model considered. The refinement and coarsening is triggered purely by flow structures. The simulation takes some time to form the characteristic vorticity structure, during which time the number of grids in the domain rises steadily. After the structure has been established, refinement and coarsening

Table 5.4: Migration statistics for the vortex street test example.

	SFC	HD
total migrations	900, 569	29, 892
individual grids migrated	28, 630	6, 870
ratio [%]	89.72	21.11
max migrations per step	378	100
average migrations per grid	37.46	4.35
max migrations per grid	331	99

equalise each other and the number of grids stays relatively constant.

Looking at the workload distribution, the relative standard deviation of grids is at its highest during the formation of the vortex street, where the number of grids in the domain is rising. After the formation of the structure, the relative standard deviation steadily decreases down to about 5% and stays at this level for the remainder of the simulation. The setup of the diffusion repartitioning also reveals a rapid progression towards a local optimum after an average of 4 diffusion steps.

The process connectivity follows the observations made for the previous examples. Applying the diffusion strategy, the total number of edges stays roughly at 1.000 with a relative standard deviation per process of around 45%, which decreases slightly to 40% in the last quarter of the simulation. The number of edges when using the SFC method stays around 600 with a relative standard deviation of 30%. The number of inter-process connections, directly between grids, is 40% higher for the diffusion method compared to the SFC. The ratio between hierarchical and spatial edges also is consistent with previous examples, with the SFC strategy exhibiting higher percentages of inter-process hierarchical edges than the diffusion-based approach.

Even though the diffusion method always reaches its local optimum in between AMR steps, the migration volume incurred by it is still considerably lower compared to the SFC method. Overall, the diffusion scheme has a migration volume of only a thirtieth of that of the SFC method.

5.4 Summary

The findings from the three test cases are summarised in the present section.

Concerning the workload balance, the diffusion approach cannot compete with the SFC method. By design, the global SFC method is able to find a optimal workload balance by simply cutting the linearised traversal of the domain tree into equally sized chunks. However, the workload balance of the diffusion approach is still viable. The diffusion methods all tend towards a relative standard deviation of less than 10 percent. With more time, the distribution of grids per process tend to become more narrow with less outliers. However, a quick rise or drop in grid numbers combined with a high spatial range of the change poses difficulties for the methods. This can be observed especially at the beginning of the testcases and at the end of the first testcase when the amount of grids starts to decrease again. The versions with no limitations on target determination perform better initially, due to more options to distribute workload to, however in time, the target determination methods geared towards a better connectivity outperform the former also in terms of workload balance. More iterations of the diffusion repartitioning, comparable to a higher order scheme, bring a slightly better balance with rapidly diminishing returns. In practice, most improvement is reached after four steps.

In terms of process connectivity, the total number of connections between processes, the diffusion repartitioning with highest degree target determination has around 40% more neighbour relations compared to the SFC partitioning. The distribution is however only slightly worse with a relative standard deviation of edges per process of roughly 45% compared to 35% for the SFC method. Additionally, the diffusion methods profit from an increase in participating processes. The SFC method on the other hand delivers only slightly worse results. The SFC is geared towards keeping favourable neighbourhood connections by distributing neighbouring grids to numerically near MPI ranks. The neighbourhood patterns slowly deteriorate when using a diffusion repartitioning. This effect is accelerated using the forced migration variants of the method, while versions without forced migration are able to keep the pattern considerably longer.

Measuring the grid connectivity, mainly important for the solution process, the diffusion method also ends up with roughly 40% more inter-process edges between grids as the SFC distribution. This ratio seems to be consistent regardless of the testcase. Furthermore, the ratios between hierarchical and spatial inter-process edges also seem to be consistent, with the SFC version exhibiting a higher number of spatial edges compared to all versions of the diffusion method. Within the diffusion method, the target determination allow to slightly influence the ratio. It cannot be determined at this point what an optimal ratio would be, which would minimise the communication volume. This is highly dependent on the solution procedure used.

By contrast, the diffusion approach exhibits considerable advantages in terms of the overall number of migrations between processes. The number of grid transfers when using an SFC distribution ranks about one to two orders of magnitude higher compared to the diffusion methods. The massive migration costs also prohibit the use of an SFC-based method for dynamic repartitioning for smaller domains with less processes. If other global methods fare better was not part of this investigation.

In conclusion, the results for workload balance and connectivity, while worse than for a SFC partitioning, are still within an acceptable range.

One caveat has to be mentioned though. In this work, neither the exact point at which global methods fail due to the effort of globally synchronising shared data structures nor their impact on the runtime has been measured extensively. Some runtime measurements have been given for the first two test cases, which show a considerable impact of the migration costs on the runtime. Nevertheless, these numbers should be taken with a grain of salt. There are a multitude of factors that influence the runtime of the simulation framework, ranging from problem parameters like the size of the individual grids, the chosen solution procedures and the frequency of the mesh adaption, as well as parameters of the repartitioning like the chosen method and the frequency in which the method is applied and finally, parameters that can hardly be influenced from the outside like the current load of a shared system among many more. Given this many options, it is hardly possible to give a fair and accurate comparison that is generally applicable. What is clear, however is that global methods will inevitably reach a point where they simply cannot be applied anymore. Therefore, the investigation into more sophisticated local repartitioning methods is certainly worthwhile and the performance seen for the investigated examples is a good starting point for future improvements.

Chapter 6

Conclusion

The rapid development of high-performance computing has enabled the tackling of some of the greatest challenges humankind faces in the 21st century. In the introduction, the historical evolution of HPC and its continued development towards more and more parallelisation has been set forth. With more parallelisation, the task to solve these challenges is not only in their mathematical modeling and the algorithmic development, but also in formulating the problem in such a way that it can make optimal use of the parallel hardware. In other words, the problem must be decomposed and distributed among all hardware elements the machine provides. Another layer of complexity is added when the decomposition changes during the solution process and the distribution is not optimal anymore. Furthermore, as these grand machines are added with more heterogeneous components for specialised tasks, a suitable distribution must take into account the strengths and weaknesses of these components.

Currently, state-of-the-art partitioning approaches use global information of the simulation domain, its decomposition and all participating processing elements. With ever growing problem sizes and machines, incorporating more and more components, this approach will not be feasible in the future. Initially, the motivation for introducing a dynamic repartitioning scheme based on a diffusion model with only local information, was to alleviate already visible bottlenecks of storing the domain structure in a central repository and updating it occasionally. During the course of this work, it has become clear that not only the partitioning is affected when problems grow, but many more components of the numerical pipeline suffer from the need to synchronise global data structures. Therefore, this work introduced a feature complete framework for the numerical simulation of real world phenomena, following the premise of limiting global data to the bare minimum wherever possible.

The basis of this framework is a completely decentralised data structure. The domain graph is generated using a hierarchical space-tree decomposition, where intermediate representations are kept. This graph is exclusively distributed among all participating actors. Each actor is only aware of its direct spatial and hierarchical neighbours. The intermediate nodes of the domain tree extend the neighbourhood model in hierarchical direction.

In addition, the grids, which more coarsely discretise the domain, are used in the solution process. Based on the data exchange requirements of all actors, several communication patterns have been introduced.

Within this work, the initial domain generation has been enhanced to make use of parallel hardware. Each process now participates in generating its own share of the data structure. Additionally, neighbourhood search and a tree balancing method, which obey the limitations of the decentral structure have been introduced. Nevertheless, an overarching macro structure is required as a starting point for domain generation. This macrostructure scales with the amount of processing units used and represents a problem that has yet to be solved.

The global communication in the input/output module has been reduced to the bare minimum. It must inevitably synchronize data twice globally. Once to determine the total size of the output file and a second time to determine the individual write offset of each processing unit.

So far, the solution procedures have not been affected by the decentral data structure. All data exchange is directly possible using the implemented communication patterns. Additionally, asynchronous methods, which look promising for decentral data structures have been added to the framework.

In order for a dynamic repartitioning to become necessary, adaptive mesh refinement and coarsening has been introduced. This includes routines for evaluating consecutive refinements and valid coarsenings across the decentral structure as well as the update of neighbourhood relationships.

The core of this thesis is the dynamic repartitioning module. It features a diffusion model dependent on the workload balance and the connectivity between neighbouring processing units to exchange workloads. Furthermore, various target determination methods were introduced. Target processes are determined based on the neighbourhood of a grid to its possible destination and its original owner.

Finally, the dynamic repartitioning has been thoroughly examined with three distinct test cases in terms of workload balance, quality of the distribution to minimise network communication and costs to reach the newly computed partitioning. The first case is an artificial benchmark, the other two were inspired by real-world applications with increasing complexity. In general, the partitioning presents itself as a viable choice. All the more, since local methods currently appear to be the only alternative if machines and simulation domains continue to grow at the current pace.

Both the concept of completely decentralised data structures and the diffusion method for partitioning have much room for improvement. Interesting choices for future developments are described in the following:

Enhanced Diffusion Model
In its respective chapter, the shortcomings of the diffusion formulation were already established. In theory, the convergence of the diffusion towards a completely balanced state is proven. In practice however, the model cannot account for load fractions and

needs to round the computed workload transfer to the nearest whole integer. This leads to either transferring too little or too much load. Furthermore, processes with already many grids tend to have a high number of neighbourhood connections. As the transfer ratio is inversely dependent on the degree (the number of neighbourhood connections), processes with a high degree tend to accumulate grids. Increasing the granularity of the decomposition might be a remedy at the expense of lower computational efficiency and higher bookkeeping costs. Another remedy might be to compute a total workload transfer per process and then migrate until this threshold has been reached. Each neighbour is assigned grids according to their relative workload transfer. Whether this solves the issue and the convergence guarantees are still valid is to be determined.

Enhanced Workload and Communication Models
Not all grids are equal when it comes to workload. Depending on the solution procedure, their refinement depth and whether they have children or are leafs, grids have various numbers of iterations to perform. As such, the computational effort a grid entails should influence the workload metric. Similarly, not each edge entails an equal amount of communication volume. Again, the same variables that influence the workload also influence edge weight. When it comes to process connectivity, a single edge might make the difference between determining neighbourhood relations between processes, which adds another layer of complexity. Again, not only should a more accurate communication model be introduced, but it should also be possible to weight the workload and communication metrics against each other. Models measuring the non-idle runtime of processes or the number of asynchronous iterations could give a very accurate estimate of the current load. Nonetheless, the insight into which factors contribute which load cannot be gathered directly. Additionally, this point is not exclusive to the diffusion partitioning. All partitioning methods need and benefit from accurate workload and communication models. Therefore, further research into these models is warranted.

Improved Target Determination
In addition to the previous point, given suitable and comparable models for workload and communication, the target determination method could be improved. That includes, but is not limited to weighing the computational cost of a grid transfer against added communication, weighing grid edges differently when they constitute the last connection to a process and would change process connectivity and migrating grids with high connectivity among them in conjunction.

Local Solution Techniques
The multigrid solver, while a very efficient technique to solve systems of linear equations, contradicts in its current form the decentralisation premise. Each multigrid cycle traverses the complete tree structure from the leafs to the root and back. In other words, the solver scales with the height of the tree and is not independent from the total size of the computational domain. The cost is very minor at the moment, because the growth ratio of the height of the tree follows a logarithmic rate compared to the growth of the resolution, still, there will be a limit to solving arbitrarily large domains in the future. As such, research into more local solution techniques is warranted. A start could be simply fixing the depth of the multigrid scheme.

Interactive Visualisation
Mundani and Frisch have introduced a visualisation technique that allows to get insight into the simulation already during runtime [229]. In an effort to limit the data transfer, the technique, called sliding window, either allows to visualise the complete domain in a coarse resolution or smaller regions with a finer resolution. The availability of coarser spatial discretisations and a central structure repository in Frisch' implementation allow this technique to be easily applied. In the present framework, coarser representations are available too, however the central repository as an entry point is missing. The hierarchy could still be traversed in an acceptable fashion directly from the process that holds the root grid and then following the children by evaluating whether they fit the visualisation window. Sending the grids to a collector for aggregation is easy, however one process cannot decide if the available transfer bandwidth to the frontend has already been reached. It is possible to get a stop signal from the collector, however this would lead to possibly unbalanced visualisations with missing data from slower or more burdened processes.

Interactive Computing
Similar to gathering data during a running simulation, the runtime connection to a framework could also be exploited to issue simulation altering tasks. This concept is called computational steering and has been explored in-depth by Mundani [228]. Possible tasks in this context could be a user-guided refinement or coarsening, altering of boundary conditions or changing other simulation parameters on the fly, and many more. It definitely is worthwhile to evaluate if and how the decentral framework concept fits into a computational steering approach.

Bibliography

[1] 102nd U.S.C., *High performance computing act of 1991, Pub.L. 102-194*, 1991, https://www.govinfo.gov/content/pkg/STATUTE-105/pdf/STATUTE-105-Pg1594.pdf.

[2] A. Abou-Rjeili and G. Karypis, *Multilevel algorithms for partitioning power-law graphs*, in Proceedings of the 20th International Conference on Parallel and Distributed Processing (IPDPS), Rhodes Island, Greece, 2006, IEEE Computer Society, p. 124. ISBN 1424400546, https://dl.acm.org/doi/10.5555/1898953.1899055.

[3] M. Ainsworth and J. Oden, *A posteriori error estimation in finite element analysis*, Computer Methods in Applied Mechanics and Engineering, 142 (1997), pp. 1–88, https://doi.org/10.1016/S0045-7825(96)01107-3.

[4] F. Allen, G. Almasi, W. Andreoni, D. Beece, B. J. Berne, A. Bright, J. Brunheroto, C. Cascaval, J. Castanos, P. Coteus, et al., *Blue gene: A vision for protein science using a petaflop supercomputer*, IBM Systems Journal, 40 (2001), pp. 310–327, https://doi.org/10.1147/sj.402.0310.

[5] S. Aluru and F. E. Sevilgen, *Parallel domain decomposition and load balancing using space-filling curves*, in Proceedings of the Fourth International Conference on High-Performance Computing, IEEE Computer Society, 1997, pp. 230–235, https://doi.org/10.1109/HIPC.1997.634498.

[6] L. Angermann, *Balanced a posteriori error estimates for finite-volume type discretizations of convection-dominated elliptic problems*, Computing, 55 (1995), pp. 305–323, https://doi.org/10.1007/BF02238485.

[7] A. Arnone, M.-S. Liou, and L. A. Povinelli, *Integration of Navier-Stokes equations using dual time stepping and a multigrid method*, AIAA journal, 33 (1995), pp. 985–990, https://doi.org/10.2514/3.12518.

[8] C. Ashcraft and J. W. Liu, *Using domain decomposition to find graph bisectors*, BIT Numerical Mathematics, 37 (1997), pp. 506–534, https://doi.org/10.1007/BF02510238.

[9] C. Aykanat, B. B. Cambazoglu, F. Findik, and T. Kurc, *Adaptive decomposition and remapping algorithms for object-space-parallel direct volume rendering*

of unstructured grids, Journal of Parallel and Distributed Computing, 67 (2007), pp. 77–99, https://doi.org/10.1016/j.jpdc.2006.05.005.

[10] M. Bader, *Space-Filling Curves: An introduction with applications in scientific computing*, vol. 9, Springer Science & Business Media, 2012. ISBN 978-3-642-31045-4.

[11] M. Bader, C. Böck, J. Schwaiger, and C. Vigh, *Dynamically adaptive simulations with minimal memory requirement—solving the shallow water equations using Sierpinski curves*, SIAM Journal on Scientific Computing, 32 (2010), pp. 212–228, https://doi.org/10.1137/080728871.

[12] M. Bader, S. Schraufstetter, C. A. Vigh, and J. Behrens, *Memory efficient adaptive mesh generation and implementation of multigrid algorithms using Sierpinski curves*, International Journal of Computational Science and Engineering, 4 (2008), pp. 12–21, https://epic.awi.de/id/eprint/16011/1/Bad2006a.pdf.

[13] P. L. Baehmann, S. L. Wittchen, M. S. Shephard, K. R. Grice, and M. A. Yerry, *Robust, geometrically based, automatic two-dimensional mesh generation*, International Journal for Numerical Methods in Engineering, 24 (1987), pp. 1043–1078, https://doi.org/10.1002/nme.1620240603.

[14] J.-L. Baer, *Microprocessor architecture: from simple pipelines to chip multiprocessors*, Cambridge University Press, 2009. ISBN 978-0521769921.

[15] J. M. Bahi, S. Contassot-Vivier, and R. Couturier, *An efficient and robust decentralized algorithm for detecting the global convergence in asynchronous iterative algorithms*, in International Conference on High Performance Computing for Computational Science, Springer, 2008, pp. 251–264, https://hal.inria.fr/inria-00336515.

[16] J. M. Bahi, S. Contassot-Vivier, R. Couturier, and F. Vernier, *A decentralized convergence detection algorithm for asynchronous parallel iterative algorithms*, IEEE Transactions on Parallel and Distributed Systems, 16 (2005), pp. 4–13, https://hal.archives-ouvertes.fr/hal-00096362/file/ConvDev_hal.pdf.

[17] S. T. Barnard, *PMRSB: Parallel multilevel recursive spectral bisection*, in Proceedings of the 1995 ACM/IEEE conference on Supercomputing, Association for Computing Machinery, 1995, pp. 27–es, https://doi.org/10.1145/224170.224227.

[18] S. T. Barnard and H. D. Simon, *Fast multilevel implementation of recursive spectral bisection for partitioning unstructured problems*, Concurrency: Practice and Experience, 6 (1994), pp. 101–117, https://doi.org/10.1002/cpe.4330060203.

[19] G. M. Baudet, *Asynchronous iterative methods for multiprocessors*, Journal of the ACM (JACM), 25 (1978), pp. 226–244, https://doi.org/10.1145/322063.322067.

[20] A. C. Bauer, *Efficient solution procedures for adaptive finite element methods: Applications to elliptic problems*, PhD thesis, State University of New York at Buffalo, 2003.

[21] M. Bauer, S. Eibl, C. Godenschwager, N. Kohl, M. Kuron, C. Rettinger, F. Schornbaum, C. Schwarzmeier, D. Thönnes, H. Köstler, et al., *walberla: A block-structured high-performance framework for multiphysics simulations*, Computers & Mathematics with Applications, 81 (2021), pp. 478–501, https://doi.org/10.1016/j.camwa.2020.01.007.

[22] D. J. Becker, T. Sterling, D. Savarese, J. E. Dorband, U. A. Ranawak, and C. V. Packer, *Beowulf: A parallel workstation for scientific computation*, in Proceedings of the International Conference on Parallel Processing, vol. 95, 1995, pp. 11–14, https://webhome.phy.duke.edu/~rgb/brahma/Resources/beowulf/papers/ICPP95/icpp95.html.

[23] M. J. Berger and S. H. Bokhari, *A partitioning strategy for nonuniform problems on multiprocessors*, IEEE Transactions on Computers, 36 (1987), pp. 570–580, https://doi.org/10.1109/TC.1987.1676942.

[24] M. J. Berger and P. Colella, *Local adaptive mesh refinement for shock hydrodynamics*, Journal of computational Physics, 82 (1989), pp. 64–84, https://doi.org/10.1016/0021-9991(89)90035-1.

[25] M. J. Berger and J. Oliger, *Adaptive mesh refinement for hyperbolic partial differential equations*, Journal of computational Physics, 53 (1984), pp. 484–512, https://doi.org/10.1016/0021-9991(84)90073-1.

[26] C. Bichot and S. C., eds., *Graph Partitioning*, Wiley, 2011, https://doi.org/10.1002/9781118601181.

[27] J. E. Boillat, *Load balancing and Poisson equation in a graph*, Concurrency: Practice and Experience, 2 (1990), pp. 289–313, https://doi.org/10.1002/cpe.4330020403.

[28] J. E. Boillat, F. Bruge, and P. Kropf, *A dynamic load-balancing algorithm for molecular dynamics simulation on multi-processor systems*, Journal of Computational Physics, 96 (1991), pp. 1–14, https://doi.org/10.1016/0021-9991(91)90263-K.

[29] E. G. Boman, Ü. V. Çatalyürek, C. Chevalier, and K. D. Devine, *The zoltan and isorropia parallel toolkits for combinatorial scientific computing: Partitioning, ordering and coloring*, Scientific Programming, 20 (2012), pp. 129–150, https://doi.org/10.3233/SPR-2012-0342.

[30] E. G. Boman, K. Devine, L. A. Fisk, R. Heaphy, B. Hendrickson, C. Vaughan, Ü. V. Çatalyürek, D. Bozdag, W. Mitchell, and J. Teresco, *Zoltan 3.0: Parallel Partitioning, Load-balancing, and Data Management Services; User's Guide*, Sandia National Laboratories, Albuquerque, NM, 2007, http://cs.sandia.gov/Zoltan/ug_html/ug.html.

[31] E. G. Boman, K. D. Devine, and S. Rajamanickam, *Scalable matrix computations on large scale-free graphs using 2d graph partitioning*, in Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis, 2013, pp. 1–12, https://doi.org/10.1145/2503210.2503293.

[32] R. B. Boppana, *Eigenvalues and graph bisection: An average-case analysis*, in 28th Annual Symposium on Foundations of Computer Science (SFCS 1987), IEEE Computer Society, 1987, pp. 280–285, https://doi.org/10.1109/SFCS.1987.22.

[33] A. Brandt, *Multi-level adaptive solutions to boundary-value problems*, Mathematics of Computation, 31 (1977), pp. 333–390, https://doi.org/10.2307/2006422.

[34] A. Breuer, A. Heinecke, and M. Bader, *Petascale local time stepping for the ader-dg finite element method*, in 2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS), IEEE Computer Society, 2016, pp. 854–863, https://doi.org/10.1109/IPDPS.2016.109.

[35] W. L. Briggs, V. E. Henson, and S. F. McCormick, *A multigrid tutorial*, SIAM, 2000. ISBN 978-0898714623.

[36] T. N. Bui and C. Jones, *A heuristic for reducing fill-in in sparse matrix factorization*, in Proceedings of the Sixth SIAM conference on Parallel Processing for Scientific Computing, SIAM, 1993, https://www.osti.gov/biblio/54439.

[37] A. Buluç, H. Meyerhenke, I. Safro, P. Sanders, and C. Schulz, *Recent advances in graph partitioning*, Algorithm engineering, (2016), pp. 117–158, https://doi.org/10.1007/978-3-319-49487-6_4.

[38] H.-J. Bungartz, M. Mehl, T. Neckel, and T. Weinzierl, *The PDE framework peano applied to fluid dynamics: an efficient implementation of a parallel multiscale fluid dynamics solver on octree-like adaptive cartesian grids*, Computational Mechanics, 46 (2010), pp. 103–114, https://doi.org/10.1007/s00466-009-0436-x.

[39] C. Burstedde, *Parallel tree algorithms for amr and non-standard data access*, ACM Transactions on Mathematical Software (TOMS), 46 (2020), pp. 1–31, https://doi.org/10.1145/3401990.

[40] C. Burstedde, J. A. Fonseca, and S. Kollet, *The simulation platform Parflow*, in JUQUEEN Extreme Scaling Workshop 2017, no. FZJ-JSC-IB-2017-01 in JSC Internal Report, Jülich Supercomputing Centre, 2017, pp. 37–42, http://hdl.handle.net/2128/13977.

[41] C. Burstedde, J. A. Fonseca, and S. Kollet, *Enhancing speed and scalability of the ParFlow simulation code*, Computational Geosciences, 22 (2018), pp. 347–361, https://doi.org/10.1007/s10596-017-9696-2.

[42] C. Burstedde, L. C. Wilcox, and O. Ghattas, *p4est: Scalable algorithms for parallel adaptive mesh refinement on forests of octrees*, SIAM Journal on Scientific Computing, 33 (2011), pp. 1103–1133, https://doi.org/10.1137/100791634.

[43] A. E. Caldwell, A. B. Kahng, and I. L. Markov, *Design and implementation of move-based heuristics for vlsi hypergraph partitioning*, Journal of Experimental Algorithmics (JEA), 5 (2000), pp. 5–es, https://doi.org/10.1145/351827.384247.

[44] K. S. Camilus and V. Govindan, *A review on graph based segmentation*, International Journal of Image, Graphics and Signal Processing, 4 (2012), p. 1, https://doi.org/10.5815/ijigsp.2012.05.01.

[45] P. M. Campbell, K. D. Devine, J. E. Flaherty, L. G. Gervasio, and J. D. Teresco, *Dynamic octree load balancing using space-filling curves*, Tech. Report CS-03-01, Williams College Department of Computer Science, 2003, https://j.teresco.org/research/publications/octpart02/octpart02.pdf.

[46] F. Cao, J. R. Gilbert, and S.-H. Teng, *Partitioning meshes with lines and planes*, tech. report, Xerox Palo Alto Research Center, 1996, https://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.56.9305&rep=rep1&type=pdf.

[47] C. Carstensen, R. Lazarov, and S. Tomov, *Explicit and averaging a posteriori error estimates for adaptive finite volume methods*, SIAM Journal on Numerical Analysis, 42 (2005), pp. 2496–2521, https://doi.org/10.1137/S0036142903425422.

[48] Ü. V. Çatalyürek and C. Aykanat, *Decomposing irregularly sparse matrices for parallel matrix-vector multiplication*, in International Workshop on Parallel Algorithms for Irregularly Structured Problems, Springer, 1996, pp. 75–86, https://faculty.cc.gatech.edu/~umit/assets/pdf/Catalyurek96.pdf.

[49] Ü. V. Çatalyürek and C. Aykanat, *Hypergraph-partitioning-based decomposition for parallel sparse-matrix vector multiplication*, IEEE Transactions on Parallel and Distributed Systems, 10 (1999), pp. 673–693, https://doi.org/10.1109/71.780863.

[50] Ü. V. Çatalyürek and C. Aykanat, *A hypergraph-partitioning approach for coarse-grain decomposition*, in SC '01: Proceedings of the 2001 ACM/IEEE Conference on Supercomputing, 2001, pp. 42–42, https://doi.org/10.1109/SC.2001.10035.

[51] Ü. V. Çatalyürek, E. G. Boman, K. D. Devine, D. Bozdag, R. Heaphy, and L. A. Riesen, *Hypergraph-based dynamic load balancing for adaptive scientific computations*, in 2007 IEEE International Parallel and Distributed Processing Symposium, IEEE Computer Society, 2007, pp. 1–11, https://doi.org/10.1109/IPDPS.2007.370258.

[52] Ü. V. Çatalyürek, E. G. Boman, K. D. Devine, D. Bozdağ, R. T. Heaphy, and L. A. Riesen, *A repartitioning hypergraph model for dynamic load balancing*, Journal of Parallel and Distributed Computing, 69 (2009), pp. 711–724, https://doi.org/10.1016/j.jpdc.2009.04.011.

[53] Ü. V. Çatalyürek, B. Uçar, and C. Aykanat, *Hypergraph partitioning*, 2011, https://doi.org/10.1007/978-0-387-09766-4_1.

[54] D. Chazan and W. Miranker, *Chaotic relaxation*, Linear Algebra and its Applications, 2 (1969), pp. 199–222, https://doi.org/10.1016/0024-3795(69)90028-7.

[55] *ChEESE: Center of excellence in solid earth*, 2021, https://cheese-coe.eu/ (accessed 2021-10-28).

[56] D.-K. Chen, H.-M. Su, and P.-C. Yew, *The impact of synchronization and granularity on parallel systems*, ACM SIGARCH Computer Architecture News, 18 (1990), pp. 239–248, https://doi.org/10.1145/325096.325150.

[57] C.-K. Cheng and Y.-C. Wei, *An improved two-way partitioning algorithm with stable performance (vlsi)*, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, 10 (1991), pp. 1502–1511, https://doi.org/10.1109/43.103500.

[58] A. Chorin, *Numerical solution of the Navier-Stokes equations*, Mathematics of Computation, 22 (1968), pp. 745–762, https://doi.org/10.2307/2004575.

[59] F. R. Chung and F. C. Graham, *Spectral graph theory*, American Mathematical Sociecty, 1997. ISBN 978-0821803158.

[60] Y.-C. Chung and S. Ranka, *Mapping finite element graphs on hypercubes*, in The Third Symposium on the Frontiers of Massively Parallel Computation, 1990, pp. 135–144, https://doi.org/10.1109/FMPC.1990.89449.

[61] D. T. Cintra, R. B. Willmersdorf, P. R. M. Lyra, and W. W. M. Lira, *A hybrid parallel dem approach with workload balancing based on hsfc*, Engineering Computations, (2016), https://doi.org/10.1108/EC-01-2016-0019.

[62] D. T. Cintra, R. B. Willmersdorf, P. R. M. Lyra, and W. W. M. Lira, *A parallel dem approach with memory access optimization using hsfc*, Engineering Computations, (2016), https://doi.org/10.1108/EC-07-2015-0203.

[63] *Chair of computational modeling and simulation*, 2022, https://www.cms.bgu.tum.de/en/ (accessed 2022-05-29).

[64] J. Cong and M. Smith, *A parallel bottom-up clustering algorithm with applications to circuit partitioning in vlsi design*, in 30th ACM/IEEE Design Automation Conference, IEEE Computer Society, 1993, pp. 755–760, https://doi.org/10.1145/157485.165119.

[65] D. Culler, J. P. Singh, and A. Gupta, *Parallel computer architecture: a hardware/software approach*, Gulf Professional Publishing, 1999. ISBN 978-1558603431.

[66] G. Cybenko, *Dynamic load balancing for distributed memory multiprocessors*, Journal of Parallel and Distributed Computing, 7 (1989), pp. 279–301, https://doi.org/10.1016/0743-7315(89)90021-X.

[67] M. Daydé, O. Marques, and K. Nakajima, eds., *Procedings of the 10th International Conference on High Performance Computing for Computational Science - VECPAR 2012*, Springer, July 2013. ISBN 978-3-642-38718-0.

[68] A. L. DeCegama, *The Technology of Parallel Processing: Parallel Processing Architectures and VLSI Hardware (Vol. 1)*, Prentice-Hall, Inc., 1989. ISBN 0139022066.

[69] H. L. deCougny, K. D. Devine, J. E. Flaherty, R. Loy, C. Özturan, and M. S. Shephard, *Load balancing for the parallel adaptive solution of partial differential equations*, Applied Numerical Mathematics, 16 (1994), pp. 157–182, https://apps.dtic.mil/sti/citations/ADA290450.

[70] D. Delling, A. V. Goldberg, T. Pajor, and R. F. Werneck, *Customizable route planning*, in Proceedings of the 10th International Symposium on Experimental Algorithms (SEA'11), Springer, 2011, pp. 376–387, https://www.microsoft.com/en-us/research/wp-content/uploads/2011/05/crp-sea.pdf.

[71] D. Delling and R. F. Werneck, *Faster customization of road networks*, in Proceedings of the 12th International Symposium on Experimental Algorithms (SEA'13), Springer, 2013, pp. 30–42, https://doi.org/10.1007/978-3-642-38527-8_5.

[72] J. W. Demmel, *Applied Numerical Linear Algebra*, SIAM, 1997. ISBN 978-0-89871-389-3.

[73] R. H. Dennard, F. H. Gaensslen, H.-N. Yu, V. L. Rideout, E. Bassous, and A. R. LeBlanc, *Design of ion-implanted mosfet's with very small physical dimensions*, IEEE Journal of Solid-State Circuits, 9 (1974), pp. 256–268, https://doi.org/10.1109/JSSC.1974.1050511.

[74] K. Devine, E. Boman, R. Heaphy, B. Hendrickson, and C. Vaughan, *Zoltan data management services for parallel dynamic applications*, Computing in Science and Engineering, 4 (2002), pp. 90–96, https://doi.org/10.1109/5992.988653.

[75] K. D. Devine, E. G. Boman, R. T. Heaphy, B. A. Hendrickson, J. D. Teresco, J. Faik, J. E. Flaherty, and L. G. Gervasio, *New challenges in dynamic load balancing*, Applied Numerical Mathematics, 52 (2005), pp. 133–152, https://dl.acm.org/doi/10.5555/1063481.1642793.

[76] K. D. Devine, E. G. Boman, V. J. Leung, L. A. Riesen, and Ü. V. Çatalyürek, *Dynamic load balancing and partitioning using the zoltan toolkit.*,

tech. report, Sandia National Lab, 2007, https://www.osti.gov/servlets/purl/1147186.

[77] K. D. Devine and J. E. Flaherty, *Parallel adaptive hp-refinement techniques for conservation laws*, Applied Numerical Mathematics, 20 (1996), pp. 367–386, https://doi.org/10.1016/0168-9274(95)00103-4.

[78] R. Diekman and R. Preis, *Load balancing strategies for distributed memory machines*, in Parallel and Distributed Processing for Computational Mechanics: Systems and Tools, Civil-Comp press, 2000, p. 124–157, https://dl.acm.org/doi/10.5555/366101.366113.

[79] R. Diekmann, D. Meyer, and B. Monien, *Parallel decomposition of unstructured fem-meshes*, in Proceedings of the Second International Workshop on Parallel Algorithms for Irregularly Structured Problems, Springer, 1995, pp. 199–215. https://doi.org/10.1002/(SICI)1096-9128(199801)10:1<53::AID-CPE288>3.0.CO;2-W.

[80] R. Diekmann, B. Monien, and R. Preis, *Using helpful sets to improve graph bisections*, Interconnection Networks and Mapping and Scheduling Parallel Computations, 21 (1995), pp. 57–73, https://doi.org/10.1090/dimacs/021.

[81] R. Diekmann, S. Muthukrishnan, and M. V. Nayakkankuppam, *Engineering diffusive load balancing algorithms using experiments*, in Proceedings of the 4th International Symposium on Solving Irregularly Structured Problems in Parallel, Springer, 1997, p. 111–122, https://dl.acm.org/doi/10.5555/646011.677011.

[82] R. Diekmann, R. Preis, F. Schlimbach, and C. Walshaw, *Shape-optimized mesh partitioning and load balancing for parallel adaptive fem*, Parallel Computing, 26 (2000), pp. 1555–1581, https://doi.org/10.1016/S0167-8191(00)00043-0.

[83] W. E. Donath and A. J. Hoffman, *Algorithms for partitioning of graphs and computer logic based on eigenvectors of connection matrices*, IBM Technical Disclosure Bulletin, 15 (1972), pp. 938–944, https://www.kde.cs.uni-kassel.de/mediawiki/images/7/70/Ibm_tdb_15-3_1972_donath-hoffman.pdf.

[84] W. E. Donath and A. J. Hoffman, *Lower bounds for the partitioning of graphs*, in Selected Papers Of Alan Hoffman: With Commentary, World Scientific, 2003, pp. 437–442, https://doi.org/10.1142/9789812796936_0044.

[85] V. Donde, V. Lopez, B. Lesieutre, A. Pinar, C. Yang, and J. Meza, *Identification of severe multiple contingencies in electric power networks*, in Proceedings of the 37th Annual North American Power Symposium, IEEE Computer Society, 2005, pp. 59–66, https://doi.org/10.1109/NAPS.2005.1560502.

[86] J. Dongarra, T. Sterling, H. Simon, and E. Strohmaier, *High-performance computing: clusters, constellations, mpps, and future directions*, Computing in Science and Engineering, 7 (2005), pp. 51–59, https://doi.org/10.1109/MCSE.2005.34.

[87] D. E. Drake and S. Hougardy, *A simple approximation algorithm for the weighted matching problem*, Information Processing Letters, 85 (2003), pp. 211–213, https://doi.org/10.1016/S0020-0190(02)00393-9.

[88] R. Duan, S. Pettie, and H.-H. Su, *Scaling algorithms for approximate and exact maximum weight matching*, Computing Research Repository, (2011), https://doi.org/10.48550/arXiv.1112.0790.

[89] A. Düster, J. Parvizian, Z. Yang, and E. Rank, *The finite cell method for three-dimensional problems of solid mechanics*, Computer Methods in Applied Mechanics and Engineering, 197 (2008), pp. 3768–3782, https://doi.org/10.1016/j.cma.2008.02.036.

[90] S. Dutt, *New faster Kernighan-Lin-type graph-partitioning algorithms*, in Proceedings of 1993 International Conference on Computer Aided Design (ICCAD), IEEE Computer Society, 1993, pp. 370–377, https://doi.org/10.1109/ICCAD.1993.580083.

[91] H. C. Edwards, *A parallel infrastructure for scalable adaptive finite element methods and its application to least squares C-infinity collocation*, PhD thesis, The University of Texas at Austin, 1997.

[92] S. Eibl and U. Rüde, *A systematic comparison of runtime load balancing algorithms for massively parallel rigid particle dynamics*, Computer Physics Communications, 244 (2019), pp. 76–85, https://doi.org/10.1016/j.cpc.2019.06.020.

[93] C. Ertl, *Scalable parallel I/O using HDF5 for hpc fluid flow simulations*, master's thesis, Technische Universität München, 2015.

[94] C. Ertl, J. Frisch, and R.-P. Mundani, *Massive parallel fluid flow simulations using hierarchical data format version 5 (HDF5)*, in 15th International Symposium on Parallel and Distributed Computing, ISPDC 2016, Fuzhou, China, 2016, https://doi.org/10.1109/ISPDC.2016.13.

[95] C. Ertl and R.-P. Mundani, *Ensuring domain consistency in an adaptive framework with distributed topology for fluid flow simulations*, in 2017 19th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC), IEEE Computer Society, 2017, pp. 312–319, https://doi.org/10.1109/SYNASC.2017.00058.

[96] E. Fadlun, R. Verzicco, P. Orlandi, and J. Mohd-Yusof, *Combined immersed-boundary finite-difference methods for three-dimensional complex flow simulations*, Journal of Computational Physics, 161 (2000), pp. 35–60, https://doi.org/10.1006/jcph.2000.6484.

[97] C. Farhat, *A simple and efficient automatic fem domain decomposer*, Computers and Structures, 28 (1988), pp. 579–602, https://doi.org/10.1016/0045-7949(88)90004-1.

[98] C. Farhat and M. Lesoinne, *Automatic partitioning of unstructured meshes for the parallel solution of problems in computational mechanics*, International Journal for Numerical Methods in Engineering, 36 (1993), pp. 745–764, https://doi.org/10.1002/nme.1620360503.

[99] M. Feldman, *Is it time for heterogeneous supercomputing?*, HPC Wire, (2006), https://www.hpcwire.com/2006/07/14/is_it_time_for_heterogeneous_supercomputing/ (accessed 2022-04-18).

[100] D. Ferrari and S. Zhou, *An empirical investigation of load indices for load balancing applications*, tech. report, University of California, Berkeley, Department of Electrical Engineering and Computer Science, 1987, https://apps.dtic.mil/sti/citations/ADA184250.

[101] J. H. Ferziger and M. Perić, *Computational methods for fluid dynamics*, vol. 3, Springer, 2002. ISBN 3319996916.

[102] C. M. Fiduccia and R. M. Mattheyses, *A linear-time heuristic for improving network partitions*, in 19th design automation conference, IEEE Computer Society, 1982, pp. 175–181, https://doi.org/10.1109/DAC.1982.1585498.

[103] M. Fiedler, *A property of eigenvectors of nonnegative symmetric matrices and its application to graph theory*, Czechoslovak Mathematical Journal, 25 (1975), pp. 619–633, http://eudml.org/doc/12900.

[104] J. Flaherty, R. Loy, M. Shephard, B. Szymanski, J. Teresco, and L. Ziantz, *Adaptive local refinement with octree load balancing for the parallel solution of three-dimensional conservation laws*, Journal of Parallel and Distributed Computing, 47 (1997), pp. 139–152, https://doi.org/https://doi.org/10.1006/jpdc.1997.1412.

[105] J. E. Flaherty, R. M. Loy, C. Özturan, M. S. Shephard, B. K. Szymanski, J. D. Teresco, and L. H. Ziantz, *Parallel structures and dynamic load balancing for adaptive finite element computation*, Applied Numerical Mathematics, 26 (1998), pp. 241–263, https://doi.org/10.1016/S0168-9274(97)00094-9.

[106] J. E. Flaherty, R. M. Loy, M. S. Shephard, B. K. Szymanski, J. D. Teresco, and L. H. Ziantz, *Adaptive local refinement with octree load balancing for the parallel solution of three-dimensional conservation laws*, Journal of parallel and distributed computing, 47 (1997), pp. 139–152, https://doi.org/10.1006/jpdc.1997.1412.

[107] J. E. Flaherty, R. M. Loy, M. S. Shephard, and J. D. Teresco, *Software for the parallel adaptive solution of conservation laws by discontinuous Galerkin methods*, in Discontinuous Galerkin Methods, Springer, 2000, pp. 113–123, https://www.osti.gov/biblio/11936.

[108] F. Fleissner and P. Eberhard, *Load balanced parallel simulation of particle-fluid dem-sph systems with moving boundaries*, Parallel Computing: Architectures, Algorithms and Applications, 48 (2007), pp. 37–44, https://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.141.1444&rep=rep1&type=pdf.

[109] F. Fleissner and P. Eberhard, *Parallel load-balanced simulation for short-range interaction particle methods with hierarchical particle grouping based on orthogonal recursive bisection*, International Journal for Numerical Methods in Engineering, 74 (2008), pp. 531–553, https://doi.org/10.1002/nme.2184.

[110] L. J. Flynn, *Intel halts development of 2 new microprocessors*, The New York Times, (2004), https://www.nytimes.com/2004/05/08/business/intel-halts-development-of-2-new-microprocessors.html (accessed 2021-08-31).

[111] M. Flynn, *Computer architecture*, Wiley Encyclopedia of Computer Science and Engineering, (2007), https://doi.org/10.1002/9780470050118.ecse071.

[112] S. Fortunato, *Community detection in graphs*, Physics reports, 486 (2010), pp. 75–174, https://doi.org/10.1016/j.physrep.2009.11.002.

[113] J. Frisch, *Towards massive parallel fluid flow simulations in computational engineering*, PhD thesis, Technische Universität München, 2014, https://mediatum.ub.tum.de/1222749?id=1222749.

[114] J. Frisch and R.-P. Mundani, *Multiskalen-Strömungssimulation eines Kraftwerkskomplexes auf Höchstleistungsrechnern [Multiscale fluid flow simulation of a power plant on high-performance computers]*, in Proceedings des 22. Forum Bauinformatik, 2010.

[115] J. Frisch and R.-P. Mundani, *Measuring and comparing the scaling behaviour of a high-performance cfd code on different supercomputing infrastructures*, in 2015 17th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC), IEEE Computer Society, 2015, pp. 371–378, https://doi.org/10.1109/SYNASC.2015.63.

[116] H. Fujii, Y. Yasuda, H. Akashi, Y. Inagami, M. Koga, O. Ishihara, M. Kashiyama, H. Wada, and T. Sumimoto, *Architecture and performance of the hitachi sr2201 massively parallel processor system*, in Proceedings 11th International Parallel Processing Symposium, IEEE Computer Society, 1997, pp. 233–241, https://doi.org/10.1109/IPPS.1997.580901.

[117] A. L. Gaitonde, *A dual-time method for the solution of the unsteady Euler equations*, The Aeronautical Journal, 98 (1994), pp. 283–291, https://doi.org/10.1017/S0001924000026786.

[118] J. Garbers, H. J. Promel, and A. Steger, *Finding clusters in vlsi circuits*, in 1990 IEEE International Conference on Computer-Aided Design, IEEE Computer Society, 1990, pp. 520–521, https://doi.org/10.1109/ICCAD.1990.129970.

[119] M. Garcia, J. Corbalan, and J. Labarta, *Lewi: A runtime balancing algorithm for nested parallelism*, in Proceedings of the International Conference on Parallel Processing. (ICPP), Sept 2009, pp. 526–533, https://doi.org/10.1109/ICPP.2009.56.

[120] M. Garcia, J. Labarta, and J. Corbalan, *Hints to improve automatic load balancing with lewi for hybrid applications*, Journal of Parallel and Distributed Computing, 74 (2014), pp. 2781–2794, https://doi.org/10.1016/j.jpdc.2014.05.004.

[121] M. R. Garey and D. S. Johnson, *Computers and Intractability; A Guide to the Theory of NP-Completeness*, W. H. Freeman & Co., 1990. ISBN 0716710455.

[122] M. R. Garey, D. S. Johnson, and L. Stockmeyer, *Some simplified np-complete problems*, in Proceedings of the Sixth Annual ACM Symposium on Theory of Computing, Association for Computing Machinery, 1974, pp. 47–63, https://doi.org/10.1145/800119.803884.

[123] A. George and J. W. Liu, *Computer Solution of Large Sparse Positive Definite*, Prentice Hall Professional Technical Reference, 1981. ISBN 0131652745.

[124] J. R. Gilbert, Y. Li, and S.-H. Teng, *Mesh partitioning toolbox – meshpart*, 2020, https://doi.org/10.5281/zenodo.3746722.

[125] J. R. Gilbert, G. L. Miller, and S.-H. Teng, *Geometric mesh partitioning: Implementation and experiments*, SIAM Journal on Scientific Computing, 19 (1998), pp. 2091–2110, https://doi.org/10.1137/S1064827594275339.

[126] J. R. Gilbert and E. Zmijewski, *A parallel graph partitioning algorithm for a message-passing multiprocessor*, International Journal of Parallel Programming, 16 (1987), pp. 427–449, https://doi.org/10.1007/BF01388998.

[127] T. Goehring and Y. Saad, *Heuristic algorithms for automatic graph partitioning*, tech. report, University of Minnesota Supercomputing Institute, 1994, https://www-users.cse.umn.edu/~saad/PDF/umsi-94-29.pdf.

[128] T. Grätsch and K.-J. Bathe, *A posteriori error estimation techniques in practical finite element analysis*, Computers & Structures, 83 (2005), pp. 235–265, https://doi.org/10.1016/j.compstruc.2004.08.011.

[129] C. Grossmann, H.-G. Roos, and M. Stynes, *Numerical treatment of partial differential equations*, vol. 154, Springer, 2007. ISBN 978-3-540-71584-9.

[130] A. Gupta, *Fast and effective algorithms for graph partitioning and sparse-matrix ordering*, IBM Journal of Research and Development, 41 (1997), pp. 171–183, https://doi.org/10.1147/rd.411.0171.

[131] W. Hackbusch, *Iterative solution of large sparse systems of equations*, vol. 95, Springer, 1994, https://doi.org/10.1007/978-3-319-28483-5.

[132] W. Hackbusch, *Multi-grid methods and applications*, vol. 4, Springer, 2013, https://doi.org/10.1007/978-3-662-02427-0.

[133] H. Hacker, C. Trinitis, J. Weidendorfer, and M. Brehm, *Considering GPGPU for HPC centers: is it worth the effort?*, Springer, 2010, pp. 118–130. ISBN 3642162320.

[134] L. Hagen and A. B. Kahng, *A new approach to effective circuit clustering*, in 1992 IEEE International Conference on Computer-Aided Design, vol. 92, 1992, pp. 422–427, https://doi.org/10.1109/ICCAD.1992.279334.

[135] W. W. Hager, S. C. Park, and T. A. Davis, *Block exchange in graph partitioning*, in Approximation and Complexity in Numerical Optimization, Springer, 2000, pp. 298–307, https://doi.org/10.1007/978-1-4757-3145-3_17.

[136] S. W. Hammond, *Mapping unstructured grid computations to massively parallel computer*, PhD thesis, Rensselaer Polytechnic Institute, Troy, New York, USA, 1992.

[137] P. Hardik, *Understanding dissipation, thermal resistance, and IC temperature*, EETimes, (2007), https://www.eetimes.com/understanding-dissipation-thermal-resistance-and-ic-temperature-part-1-of-2/ (accessed 2021-08-31).

[138] L. Hart and S. McCormick, *Asynchronous multilevel adaptive methods for solving partial differential equations on multiprocessors: Basic ideas*, Parallel Computing, 12 (1989), pp. 131–144, https://doi.org/10.1016/0167-8191(89)90048-3.

[139] S. Hauck and G. Borriello, *An evaluation of bipartitioning techniques*, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, 16 (1997), pp. 849–866, https://doi.org/10.1109/43.644609.

[140] M. T. Heath and P. Raghavan, *A cartesian parallel nested dissection algorithm*, SIAM Journal on Matrix Analysis and Applications, 16 (1995), pp. 235–253, https://doi.org/10.1137/S0895479892238270.

[141] B. Hendrickson and K. Devine, *Dynamic load balancing in computational mechanics*, Computer methods in applied mechanics and engineering, 184 (2000), pp. 485–500, https://doi.org/10.1016/S0045-7825(99)00241-8.

[142] B. Hendrickson and T. G. Kolda, *Graph partitioning models for parallel computing*, Parallel computing, 26 (2000), pp. 1519–1534, https://doi.org/10.1016/S0167-8191(00)00048-X.

[143] B. Hendrickson and R. Leland, *The chaco user's guide: Version 2.0*, tech. report, Sandia National Laboratories, 1995, https://cfwebprod.sandia.gov/cfdocs/CompResearch/docs/guide.pdf.

[144] B. Hendrickson and R. Leland, *An improved spectral graph partitioning algorithm for mapping parallel computations*, SIAM Journal on Scientific Computing, 16 (1995), pp. 452–469, https://doi.org/10.1137/0916028.

[145] B. Hendrickson and R. Leland, *A multi-level algorithm for partitioning graphs*, in Supercomputing '95: Proceedings of the 1995 ACM/IEEE Conference on Supercomputing, Association for Computing Machinery, 1995, pp. 28–28, https://doi.org/10.1145/224170.224228.

[146] B. Hendrickson, R. Leland, and R. Van Driessche, *Skewed graph partitioning*, tech. report, Sandia National Labs, 1997, https://cfwebprod.sandia.gov/cfdocs/CompResearch/docs/skew.pdf.

[147] J. L. Hennessy and D. A. Patterson, *Computer architecture: a quantitative approach*, Elsevier, 2011. ISBN 978-8178672663.

[148] C. Herzfeld, *Grand challenges 1992, high performance computing and communications, the FY 1992 U.S. research and development program*, A Report by the Committee on Physical, Mathematical, and Engineering Sciences, (1991), https://www.nitrd.gov/pubs/bluebooks/1992/pdf/bluebook92.pdf.

[149] D. Hilbert, *Über die stetige Abbildung einer Linie auf ein Flächenstück*, in Dritter Band: Analysis · Grundlagen der Mathematik · Physik Verschiedenes, Springer, 1935, pp. 1–2. ISBN 978-3-662-37657-7.

[150] M. D. Hill, N. P. Jouppi, , and G. S. Sohi, *Readings in computer architecture*, Morgan Kaufmann, 2000. ISBN 978-1558605398.

[151] N. Hirose and M. Fukuda, *Numerical wind tunnel (nwt) and CFD research at national aerospace laboratory*, in Proceedings High Performance Computing on the Information Superhighway. HPC Asia'97, IEEE Computer Society, 1997, pp. 99–103, https://doi.org/10.1109/HPC.1997.592130.

[152] A. R. Hoffman, *Supercomputers: directions in technology and applications*, National Academies Press, 1989, https://doi.org/10.17226/1405.

[153] M. Holtgrewe, P. Sanders, and C. Schulz, *Engineering a scalable high quality graph partitioner*, in 2010 IEEE International Symposium on Parallel and Distributed Processing (IPDPS), IEEE Computer Society, 2010, pp. 1–12, https://doi.org/10.1109/IPDPS.2010.5470485.

[154] R. M. Hord, *The Illiac IV: the first supercomputer*, Springer, 2013. ISBN 978-3540117650.

[155] G. Horton, *A multi-level diffusion method for dynamic load balancing*, Parallel Computing, 19 (1993), pp. 209–218, https://doi.org/10.1016/0167-8191(93)90050-U.

[156] G. Houzeaux, M. Garcia, J. C. Cajas, A. Artigues, E. Olivares, J. Labarta, and M. Vázquez, *Dynamic load balance applied to particle transport in fluids*, International Journal of Computational Fluid Dynamics, 30 (2016), pp. 408–418, https://doi.org/10.1080/10618562.2016.1227070.

[157] G. Houzeaux, M. Garcia-Gasulla, J. Cajas, R. Borrell, A. Santiago, C. Moulinec, and M. Vázquez, *Parallel multiphysics coupling: Algorithmic and computational performances*, International Journal of Computational Fluid Dynamics, 34 (2020), pp. 486–507, https://doi.org/10.1080/10618562.2020.1783440.

[158] G. Houzeaux, M. Vázquez, R. Aubry, and J. M. Cela, *A massively parallel fractional step solver for incompressible flows*, Journal of Computational Physics, 228 (2009), pp. 6316–6332, https://doi.org/10.1016/j.jcp.2009.05.019.

[159] J. Hromkovič and B. Monien, *The bisection problem for graphs of degree 4 (configuring transputer systems)*, in International Symposium on Mathematical Foundations of Computer Science, Springer, 1991, pp. 211–220, https://doi.org/10.1007/3-540-54345-7_64.

[160] S.-H. Hsieh, G. H. Paulino, and J. F. Abel, *Evaluation of automatic domain partitioning algorithms for parallel finite element analysis*, International Journal for Numerical Methods in Engineering, 40 (1997), pp. 1025–1051. https://doi.org/10.1002/(SICI)1097-0207(19970330)40:6<1025::AID-NME103>3.0.CO;2-P.

[161] Y. F. Hu and R. J. Blake, *An optimal dynamic load balancing algorithm*, tech. report, Daresbury Laboratory, 1995, http://citeseerx.ist.psu.edu/viewdoc/download;jsessionid=464A8F41B711490D2E90B898A3432135?doi=10.1.1.51.4963&rep=rep1&type=pdf.

[162] K. Hwang and A. Faye, *Computer architecture and parallel processing*, McGraw-Hill, 1984. ISBN 978-0070315563.

[163] K. Hwang and N. Jotwani, *Advanced computer architecture: parallelism, scalability, programmability*, McGraw-Hill, 1993. ISBN 978-0070316225.

[164] T. Isaac, C. Burstedde, and O. Ghattas, *Low-cost parallel algorithms for 2:1 octree balance*, in 2012 IEEE 26th International Parallel and Distributed Processing Symposium, IEEE Computer Society, 2012, pp. 426–437, https://doi.org/10.1109/IPDPS.2012.47.

[165] T. Isaac, C. Burstedde, L. C. Wilcox, and O. Ghattas, *Recursive algorithms for distributed forests of octrees*, SIAM Journal on Scientific Computing, 37 (2015), pp. C497–C531, https://doi.org/10.1137/140970963.

[166] G. J. Jacobson, *Succinct static data structures*, PhD thesis, Carnegie Mellon University, 1988.

[167] H. V. Jagadish, *Linear clustering of objects with multiple attributes*, Proceedings of the 1990 ACM SIGMOD International Conference on Management of Data, 19 (1990), pp. 332–342, https://doi.org/10.1145/93605.98742.

[168] E. Jeannot and F. Vernier, *A practical approach of diffusion load balancing algorithms*, in European Conference on Parallel Processing, Springer, 2006, pp. 211–221, https://doi.org/10.1007/11823285_22.

[169] D. Johnson, *IBM introduces the world's first 2-nm node chip*, IEEE Spectrum, (2021), https://spectrum.ieee.org/ibm-introduces-the-worlds-first-2nm-node-chip (accessed 2021-08-30).

[170] J. N. Jomo, *Towards scalable finite cell computations on massively parallel systems*, PhD thesis, Technische Universität München, 2021, http://nbn-resolving.de/urn/resolver.pl?urn:nbn:de:bvb:91-diss-20210426-1576808-1-9.

[171] J. N. Jomo, N. Zander, M. Elhaddad, A. Özcan, S. Kollmannsberger, R.-P. Mundani, and E. Rank, *Parallelization of the multi-level hp-adaptive finite cell method*, Computers & Mathematics with Applications, 74 (2017), pp. 126–142, https://doi.org/10.1016/j.camwa.2017.01.004. 5th European Seminar on Computing ESCO 2016.

[172] M. T. Jones and P. E. Plassmann, *Computational results for parallel unstructured mesh computations*, Computing Systems in Engineering, 5 (1994), pp. 297–309.

[173] B. H. Junker and F. Schreiber, *Analysis of biological networks*, vol. 2, Wiley, 2011, https://doi.org/10.1002/9780470253489.

[174] G. Karypis, *Metis: Unstructured graph partitioning and sparse matrix ordering system*, tech. report, Department of Computer Science, University of Minnesota, 1997, https://dm.kaist.ac.kr/kse625/resources/metis.pdf.

[175] G. Karypis, R. Aggarwal, V. Kumar, and S. Shekhar, *Multilevel hypergraph partitioning: Applications in vlsi domain*, IEEE Transactions on Very Large Scale Integration (VLSI) Systems, 7 (1999), pp. 69–79, https://doi.org/10.1109/92.748202.

[176] G. Karypis and V. Kumar, *Analysis of multilevel graph partitioning*, in Supercomputing'95: Proceedings of the 1995 ACM/IEEE conference on Supercomputing, Association for Computing Machinery, 1995, pp. 29–29, https://doi.org/10.1145/224170.224229.

[177] G. Karypis and V. Kumar, *A coarse-grain parallel formulation of multilevel k-way graph partitioning algorithm*, in Proceedings of the Eighth SIAM conference on Parallel Processing for Scientific Computing, 1997, http://glaros.dtc.umn.edu/gkhome/node/84.

[178] G. Karypis and V. Kumar, *A fast and high quality multilevel scheme for partitioning irregular graphs*, SIAM Journal on Scientific Computing, 20 (1998), pp. 359–392, http://glaros.dtc.umn.edu/gkhome/node/107.

[179] G. Karypis and V. Kumar, *Multilevel k-way partitioning scheme for irregular graphs*, Journal of Parallel and Distributed Computing, 48 (1998), pp. 96–129, http://glaros.dtc.umn.edu/gkhome/node/81.

[180] G. Karypis and V. Kumar, *Multilevel k-way partitioning scheme for irregular graphs*, Journal of Parallel and Distributed Computing, 48 (1998), pp. 96–129, https://doi.org/10.1006/jpdc.1997.1404.

[181] G. Karypis, K. Schloegel, and V. Kumar, *Parmetis: Parallel graph partitioning and sparse matrix ordering library*, tech. report, Department of Computer Science, University of Minnesota, 1997, http://glaros.dtc.umn.edu/gkhome/fetch/sw/parmetis/manual.pdf.

[182] D. Kelly, R. Mills, J. Reizes, and A. Miller, *A posteriori error estimates in finite difference techniques*, Journal of Computational Physics, 74 (1988), pp. 214–232, https://doi.org/10.1016/0021-9991(88)90077-0.

[183] B. W. Kernighan and S. Lin, *An efficient heuristic procedure for partitioning graphs*, The Bell System Technical Journal, 49 (1970), pp. 291–307, https://doi.org/10.1002/j.1538-7305.1970.tb01770.x.

[184] T. Kieritz, D. Luxen, P. Sanders, and C. Vetter, *Distributed time-dependent contraction hierarchies*, in International Symposium on Experimental Algorithms, Springer, 2010, pp. 83–93, https://doi.org/10.1007/978-3-642-13193-6_8.

[185] J. Kim, D. Kim, and H. Choi, *An immersed-boundary finite-volume method for simulations of flow in complex geometries*, Journal of Computational Physics, 171 (2001), pp. 132–150, https://doi.org/10.1006/jcph.2001.6778.

[186] S. Kollmannsberger and P. Kopp, *On accurate time integration for temperature evolutions in additive manufacturing*, GAMM-Mitteilungen, 44 (2021), https://doi.org/10.1002/gamm.202100019.

[187] P. Kopp, V. Calo, E. Rank, and S. Kollmannsberger, *Space-time hp finite elements for heat evolution in laser-based additive manufacturing*, 2021, https://arxiv.org/abs/2112.00155.

[188] P. Kopp, E. Rank, V. M. Calo, and S. Kollmannsberger, *Efficient multi-level hp-finite elements in arbitrary dimensions*, 2021, https://arxiv.org/abs/2106.08214.

[189] Q. Koziol, *HDF5*, in Encyclopedia of Parallel Computing, D. Padua, ed., Springer, 2011, pp. 827–833, https://doi.org/10.1007/978-0-387-09766-4_44.

[190] B. Kruatrachue and T. Lewis, *Grain size determination for parallel processing*, IEEE Software, 5 (1988), pp. 23–32, https://doi.org/10.1109/52.1991.

[191] C. Lanczos, *An iteration method for the solution of the eigenvalue problem of linear differential and integral operators*, Journal of Research of the National Bureau of Standards, 45 (1950), pp. 255–282, https://nvlpubs.nist.gov/nistpubs/jres/045/jresv45n4p255_a1b.pdf.

[192] U. Lauther, *An extremely fast, exact algorithm for finding shortest paths in static networks with geographical background*, Proceedings Münster GI-Days, Konferenz für Geoinformatik in Münster, (2006), https://studylib.net/doc/18221275/an-extremely-fast--exact-algorithm-for-finding-shortest.

[193] H. Lebesgue, *Leçons sur l'intégration et la recherche des fonctions primitives*, Gauthier-Villars, 1904, https://gallica.bnf.fr/ark:/12148/bpt6k377935x.texteImage.

[194] B. Lee, S. F. McCormick, B. Philip, and D. J. Quinlan, *Asynchronous fast adaptive composite-grid methods: Numerical results*, SIAM journal on Scientific Computing, 25 (2003), pp. 682–700, https://doi.org/10.1137/S1064827502407536.

[195] B. Lee, S. F. McCormick, B. Philip, and D. J. Quinlan, *Asynchronous fast adaptive composite-grid methods for elliptic problems: Theoretical foundations*, SIAM Journal on Numerical Analysis, 42 (2004), pp. 130–152, https://doi.org/10.1137/S0036142902400767.

[196] H. Lee, L.-E. Yu, S.-W. Ryu, J.-W. Han, K. Jeon, D.-Y. Jang, K.-H. Kim, J. Lee, J.-H. Kim, S. Jeon, et al., *Sub-5nm all-around gate finfet for ultimate scaling*, in 2006 Symposium on VLSI Technology, 2006. Digest of Technical Papers, IEEE Computer Society, 2006, pp. 58–59, https://doi.org/10.1109/VLSIT.2006.1705215.

[197] E. Leiss and H. Reddy, *Distributed load balancing: design and performance analysis*, WM Keck Research Computation Laboratory, 5 (1989), pp. 205–270.

[198] L. E. Levine and B. Lane, *2018 am-bench test descriptions for amb2018-02*, 2019, https://www.nist.gov/ambench/amb2018-02-description (accessed 2022-03-02).

[199] J. Li and C.-C. Liu, *Power system reconfiguration based on multilevel graph partitioning*, in 2009 IEEE Bucharest PowerTech, IEEE Computer Society, 2009, pp. 1–5, https://doi.org/10.1109/PTC.2009.5281971.

[200] *LRZ CoolMUC-2 documentation*, 2021, https://doku.lrz.de/display/PUBLIC/CoolMUC-2 (accessed 2021/02/17).

[201] *LRZ Linux Cluster documentation*, 2021, https://doku.lrz.de/display/PUBLIC/Linux+Cluster (accessed 2021/02/17).

[202] D. Luxen and D. Schieferdecker, *Candidate sets for alternative routes in road networks*, Journal of Experimental Algorithmics (JEA), 19 (2015), pp. 1–28, https://doi.org/10.1145/2674395.

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

[204] D. Markauskas and A. Kačeniauskas, *The comparison of two domain repartitioning methods used for parallel discrete element computations of the hopper discharge*, Advances in Engineering Software, 84 (2015), pp. 68–76, https://doi.org/10.1016/j.advengsoft.2014.12.002.

[205] W. Massey, *Grand challenges 1993, high performance computing and communications, the FY 1993 U.S. research and development program*, A Report by the Committee on Physical, Mathematical, and Engineering Sciences, (1992), https://www.nitrd.gov/pubs/bluebooks/1993/pdf/bluebook93.pdf.

[206] T. G. Mattson, *An overview of the intel tflops supercomputer*, Intel Technology Journal, (1998), http://www.ai.mit.edu/projects/aries/course/notes/ascii_red.pdf.

[207] J. Maue and P. Sanders, *Engineering algorithms for approximate weighted matching*, in International Workshop on Experimental and Efficient Algorithms, Springer, 2007, pp. 242–255, https://doi.org/10.1007/978-3-540-72845-0_19.

[208] S. McCormick and D. Quinlan, *Asynchronous multilevel adaptive methods for solving partial differential equations on multiprocessors: Performance results*, Parallel Computing, 12 (1989), pp. 145–156, https://doi.org/10.1016/0167-8191(89)90049-5.

[209] C. McCreary and H. Gill, *Automatic determination of grain size for efficient parallel processing*, Communications of the ACM, 32 (1989), pp. 1073–1078, https://doi.org/10.1145/66451.66454.

[210] A. Meister, *Numerik linearer Gleichungssysteme*, vol. 4, Springer, 2011. ISBN 978-3-658-07199-8.

[211] O. Meister, *Sierpinski Curves for Parallel Adaptive Mesh Refinement in Finite Element and Finite Volume Methods*, PhD thesis, Technische Universität München, 2016.

[212] H. Meyerhenke, B. Monien, and S. Schamberger, *Accelerating shape optimizing load balancing for parallel fem simulations by algebraic multigrid*, in Proceedings 20th IEEE International Parallel & Distributed Processing Symposium, IEEE Computer Society, 2006, pp. 10–pp, https://doi.org/10.1109/IPDPS.2006.1639295.

[213] H. Meyerhenke and S. Schamberger, *Balancing parallel adaptive fem computations by solving systems of linear equations*, in 11th International Euro-Par Conference, Springer, 2005, pp. 209–219, http://parco.iti.uka.de/henningm/data/europar05.pdf.

[214] G. L. Miller, S.-H. Teng, W. Thurston, and S. A. Vavasis, *Automatic mesh partitioning*, in Graph Theory and Sparse Matrix Computation, Springer, 1993, pp. 57–84, https://www.cs.cmu.edu/~glmiller/Publications/b2hd-MiTeThVa93.html.

[215] G. L. Miller, S.-H. Teng, and S. A. Vavasis, *Unified geometric approach to graph separators*, in Proceedings of the 32nd Symposium on Foundations of Computer Science, IEEE Computer Society, 1991, pp. 538–547, https://doi.org/10.1109/SFCS.1991.185417.

[216] T. Minyard and Y. Kallinderis, *Octree partitioning of hybrid grids for parallel adaptive viscous flow simulations*, International journal for numerical methods in fluids, 26 (1998), pp. 57–78. https://doi.org/10.1002/(SICI)1097-0363(19980115)26:1<57::AID-FLD625>3.0.CO;2-N.

[217] T. Minyard and Y. Kallinderis, *Parallel load balancing for dynamic execution environments*, Computer Methods in Applied Mechanics and Engineering, 189 (2000), pp. 1295–1309, https://doi.org/10.2514/6.1996-295.

[218] R. H. Möhring, H. Schilling, B. Schütz, D. Wagner, and T. Willhalm, *Partitioning graphs to speedup dijkstra's algorithm*, Journal of Experimental Algorithmics (JEA), 11 (2007), pp. 2–8, https://doi.org/10.1145/1187436.1216585.

[219] B. Monien, R. Preis, and R. Diekmann, *Quality matching and local improvement for multilevel graph-partitioning*, Parallel Computing, 26 (2000), pp. 1609–1634, https://doi.org/10.1016/S0167-8191(00)00049-1.

[220] B. Monien and S. Schamberger, *Graph partitioning with the party library: Helpful-sets in practice*, in 16th Symposium on Computer Architecture and High Performance Computing, IEEE Computer Society, 2004, pp. 198–205, https://doi.org/10.1109/SBAC-PAD.2004.18.

[221] G. E. Moore, *Cramming more components onto integrated circuits*, Electronics, 38 (1965), p. 114ff, https://newsroom.intel.com/wp-content/uploads/sites/11/2018/05/moores-law-electronics.pdf.

[222] G. E. Moore, *Progress in digital integrated electronics*, in International Electron devices meeting, vol. 21, IEEE Computer Society, 1975, pp. 11–13, https://www.eng.auburn.edu/~agrawvd/COURSE/E7770_Spr07/READ/Gordon_Moore_1975_Speech.pdf.

[223] J. E. Moreira, D. Schouten, and C. Polychronopoulos, *The performance impact of granularity control and functional parallelism*, in Proceedings of the 8th International Workshop on Languages and Compilers for Parallel Computing, Springer, 1995, pp. 581–597.

[224] G. M. Morton, *A computer oriented geodetic data base and a new technique in file sequencing*, tech. report, IBM, 1966, https://dominoweb.draco.res.ibm.com/0dabf9473b9c86d48525779800566a39.html.

[225] *MPI: A message-passing interface standard, version 4.0*, tech. report, Message Passing Interface Forum, June 2021, https://www.mpi-forum.org/docs/mpi-4.0/mpi40-report.pdf.

[226] R.-P. Mundani, *Hierarchische Geometriemodelle zur Einbettung verteilter Simulationsaufgaben*, dissertation, Technische Universität München, München, 2006.

[227] R.-P. Mundani, *Hierarchische Geometriemodelle zur Einbettung verteilter Simulationsaufgaben [Hierarchical Geometry Models to Embed Distributed Simulation Tasks]*, Shaker, 2006. ISBN 978-3832249069.

[228] R.-P. Mundani, *Interaction in High Performance Computing Scenarios*, PhD thesis, Technische Universität München, 2014.

[229] R.-P. Mundani, J. Frisch, V. Varduhn, and E. Rank, *A sliding window technique for interactive high-performance computing scenarios*, Advances in Engineering Software, 84 (2015), pp. 21–30, https://doi.org/10.1016/j.advengsoft.2015.02.003.

[230] C. J. Murray, *The supermen: The story of Seymour Cray and the technical wizards behind the supercomputer*, Wiley, 1997. ISBN 978-0471048855.

[231] S. P. Muszala, G. Alaghband, J. Hack, and D. Connors, *Natural load indices (nli) for scientific simulation*, The Journal of Supercomputing, 59 (2012), pp. 392–413, https://doi.org/10.1007/s11227-010-0442-y.

[232] J. T. Oden, A. Patra, and Y. Feng, *Parallel domain decomposition solver for adaptive hp finite element methods*, SIAM Journal on Numerical Analysis, 34 (1997), pp. 2090–2118, https://doi.org/10.1137/S0036142994278887.

[233] V. Osipov and P. Sanders, *n-level graph partitioning*, in European Symposium on Algorithms, Springer, 2010, pp. 278–289, https://doi.org/10.48550/arXiv.1004.4024.

[234] C. Özturan, *Distributed environment and load balancing for adaptive unstructured meshes.*, PhD thesis, Computer Science Department, Rensselaer Polytechnic Institute, 1997.

[235] M. Parashar and J. C. Browne, *On partitioning dynamic adaptive grid hierarchies*, in Proceedings of HICSS-29: 29th Hawaii International Conference on System Sciences, vol. 1, IEEE Computer Society, 1996, pp. 604–613, https://doi.org/10.1109/HICSS.1996.495511.

[236] M. Parashar, J. C. Browne, C. Edwards, and K. Klimkowski, *A common data management infrastructure for adaptive algorithms for pde solutions*, in SC'97: Proceedings of the 1997 ACM/IEEE Conference on Supercomputing, IEEE Computer Society, 1997, pp. 56–56, https://doi.org/10.1145/509593.509649.

[237] *ParFlow hydrologic model: Modelling surface and subsurface flow on high-performance computers*, 2021, https://parflow.org/ (accessed 2021-10-29).

[238] B. N. Parlett, H. Simon, and L. Stringer, *On estimating the largest eigenvalue with the lanczos algorithm*, Mathematics of Computation, 38 (1982), pp. 153–165, https://doi.org/10.2307/2007471.

[239] J. Parvizian, A. Düster, and E. Rank, *Finite cell method*, Computational Mechanics, 41 (2007), pp. 121–133, https://doi.org/10.1007/s00466-007-0173-y.

[240] A. Patra and J. T. Oden, *Problem decomposition for adaptive hp finite element methods*, Computing Systems in Engineering, 6 (1995), pp. 97–109, https://doi.org/10.1016/0956-0521(95)00008-N.

[241] G. Peano, *Sur une courbe, qui remplit toute une aire plane*, Mathematische Annalen, 36 (1890), pp. 157–160, https://doi.org/10.1007/BF01199438.

[242] F. Pellegrini, *Pt-scotch and libscotch 6.1 user's guide*, tech. report, INRIA, 2021, https://hal.archives-ouvertes.fr/hal-00410328/document.

[243] F. Pellegrini and J. Roman, *Scotch: A software package for static mapping by dual recursive bipartitioning of process and architecture graphs*, in International Conference on High-Performance Computing and Networking, Springer, 1996, pp. 493–498, https://doi.org/10.1007/3-540-61142-8_588.

[244] B. Peng, L. Zhang, and D. Zhang, *A survey of graph theoretical approaches to image segmentation*, Pattern recognition, 46 (2013), pp. 1020–1038, https://doi.org/10.1016/j.patcog.2012.09.015.

[245] N. Perovic-Rott, *Coupling of Models and Scales in Computational Fluid Dynamics*, PhD thesis, Technische Universität München, 2021, https://mediatum.ub.tum.de/1554517.

[246] A. Petitet, R. C. Whalex, J. Dongarra, and A. Cleary, *HPL - a portable implementation of the high-performance linpack benchmark for distributed-memory computers*, 2018, http://www.netlib.org/benchmark/hpl/ (accessed 2021-08-06).

[247] S. Pettie and P. Sanders, *A simpler linear time 2/3- ε approximation for maximum weight matching*, Information processing letters, 91 (2004), pp. 271–276, https://web.eecs.umich.edu/~pettie/papers/twothirds.pdf.

[248] M. Pfaffinger, *Interaktive Strömungssimulation auf Hochleistungsrechnern unter Anwendung der Lattice-Boltzmann Methode [Interactive Fluid Flow Simulation on High-Performance Computers using the Lattice-Boltzmann Method*, PhD thesis, Technische Universität München, 2012.

[249] J. R. Pilkington and S. B. Baden, *Partitioning with spacefilling curves*, tech. report, Department of Computer Science and Engineering, University of California, 1994, http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.31.847&rep=rep1&type=pdf.

[250] J. R. Pilkington and S. B. Baden, *Dynamic partitioning of non-uniform structured workloads with spacefilling curves*, IEEE Transactions on parallel and Distributed Systems, 7 (1996), pp. 288–300, https://doi.org/10.1109/71.491582.

[251] S. Plimpton, S. Attaway, B. Hendrickson, J. Swegle, C. Vaughan, and D. Gardner, *Parallel transient dynamics simulations: Algorithms for contact detection and smoothed particle hydrodynamics*, Journal of Parallel and Distributed Computing, 50 (1998), pp. 104–122, https://doi.org/10.1145/369028.369087.

[252] A. Pothen, H. D. Simon, and K.-P. Liou, *Partitioning sparse matrices with eigenvectors of graphs*, SIAM Journal on Matrix Analysis and Applications, 11 (1990), pp. 430–452, `https://doi.org/10.1137/0611030`.

[253] A. Pothen, H. D. Simon, L. Wang, and S. T. Barnard, *Towards a fast implementation of spectral nested dissection*, in Supercomputing'92: Proceedings of the 1992 ACM/IEEE Conference on Supercomputing, IEEE Computer Society, 1992, pp. 42–51, `https://doi.org/10.1109/SUPERC.1992.236711`.

[254] R. Preis, *Linear time 1/2-approximation algorithm for maximum weighted matching in general graphs*, in Annual Symposium on Theoretical Aspects of Computer Science, Springer, 1999, pp. 259–269, `https://dl.acm.org/doi/10.5555/1764891.1764924`.

[255] R. Preis and R. Diekmann, *Party-a software library for graph partitioning*, Advances in Computational Mechanics with Parallel and Distributed Processing, (1997), pp. 63–71, `https://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.50.9217&rep=rep1&type=pdf`.

[256] T. Pricket Morgan, *Top 500 supers – the dawning of the GPUs*, The Register, (2010), `https://www.theregister.com/2010/05/31/top_500_supers_jun2010/` (accessed 2021-09-07).

[257] J. M. Rabaey and M. Pedram, *Low power design methodologies*, vol. 336, Springer, 2012, `https://doi.org/10.1007/978-1-4615-2307-9`.

[258] E. Rank, *A-posteriori-Fehlerabschätzungen und adaptive Netzverfeinernung für Finite-Element- und Randintegralelement-Methoden*, PhD thesis, Technische Universität München, 1984, `https://mediatum.ub.tum.de/1457349`.

[259] E. Rank, *Adaptivity and accuracy estimation for finite element and boundary integral element methods*, Accuracy estimates and adaptive refinements in finite element computations, (1986), pp. 79–94, `http://www.cie.bgu.tum.de/publications/proceedings/1985_05.pdf`.

[260] E. Rank, *Adaptive remeshing and h-p domain decomposition*, Computer Methods in Applied Mechanics and Engineering, 101 (1992), pp. 299–313, `https://doi.org/10.1016/0045-7825(92)90027-H`.

[261] A. Reinarz, D. E. Charrier, M. Bader, L. Bovard, M. Dumbser, K. Duru, F. Fambri, A.-A. Gabriel, J.-M. Gallard, S. Köppel, L. Krenz, L. Rannabauer, L. Rezzolla, P. Samfass, M. Tavelli, and T. Weinzierl, *Exahype: An engine for parallel dynamically adaptive simulations of wave problems*, Computer Physics Communications, 254 (2020), p. 107251, `https://doi.org/10.1016/j.cpc.2020.107251`.

[262] O. Rivera and M. Käser, *Toward Optimal Load Balance in Parallel Programs for Geophysical Simulations*, InSiDE, 6 (2008), pp. 46–49.

[263] P. Sadayappan and F. Ercal, *Nearest-neighbor mapping of finite element graphs onto processor meshes*, IEEE Transactions on Computers, 100 (1987), pp. 1408–1424, https://doi.org/10.1109/TC.1987.5009494.

[264] I. Safro, P. Sanders, and C. Schulz, *Advanced coarsening schemes for graph partitioning*, Journal of Experimental Algorithmics (JEA), 19 (2015), pp. 1–24, https://doi.org/10.1145/2670338.

[265] H. Sagan, *Space-filling curves*, Springer, 2012, https://doi.org/10.1007/978-1-4612-0871-6.

[266] S. Salihoglu and J. Widom, *Gps: A graph processing system*, in Proceedings of the 25th International Conference on Scientific and Statistical Database Management, Association for Computing Machinery, 2013, pp. 1–12, https://doi.org/10.1145/2484838.2484843.

[267] H. Samet, *The design and analysis of spatial data structures*, vol. 85, Addison Wesley, 1990. ISBN 978-0201502558.

[268] *sam(oa)^2 – space filling curves and adaptive meshes for oceanic and other applications*, 2021, http://www.github.com/meistero/samoa (accessed 2021-10-29).

[269] P. Sanders and C. Schulz, *Engineering multilevel graph partitioning algorithms*, in European Symposium on Algorithms, Springer, 2011, pp. 469–480, https://doi.org/10.1007/978-3-642-23719-5_40.

[270] P. Sanders and C. Schulz, *Kahip–karlsruhe high qualtity partitioning homepage*, 2013, http://algo2.iti.kit.edu/documents/kahip/index.html (accessed 2021-10-9).

[271] P. Sanders and C. Schulz, *Think locally, act globally: Highly balanced graph partitioning*, in International Symposium on Experimental Algorithms, Springer, 2013, pp. 164–175, https://doi.org/10.1007/978-3-642-38527-8_16.

[272] B. Santo, *Intel charts manufaturing course to 2025*, EETimes, (2021), https://www.eetimes.com/intel-charts-manufacturing-course-to-2025/ (accessed 2021-08-30).

[273] M. Schäfer, S. Turek, F. Durst, E. Krause, and R. Rannacher, *Benchmark computations of laminar flow around a cylinder*, in Flow Simulation with High-Performance Computers II, vol. 52 of Notes on Numerical Fluid Mechanics, Vieweg, 1996, pp. 547—-566, https://doi.org/10.1007/978-3-322-89849-4_39.

[274] S. Schamberger, *On partitioning fem graphs using diffusion*, in 18th International Parallel and Distributed Processing Symposium, 2004. Proceedings., IEEE Computer Society, 2004, p. 277, https://doi.org/10.1109/IPDPS.2004.1303358.

[275] D. Schillinger, *The p- and B-spline versions of the geometrically nonlinear finite cell method and hierarchical refinement strategies for adaptive isogeometric and embedded domain analysis*, PhD thesis, Technische Universität München, 2011, https://mediatum.ub.tum.de/1093297.

[276] D. Schillinger and M. Ruess, *The finite cell method: A review in the context of higher-order structural analysis of CAD and image-based geometric models*, Archives of Computational Methods in Engineering, 22 (2015), pp. 391–455, https://doi.org/10.1007/s11831-014-9115-y.

[277] K. Schloegel, G. Karypis, and V. Kumar, *Multilevel diffusion schemes for repartitioning of adaptive meshes*, Journal of Parallel and Distributed Computing, 47 (1997), pp. 109–124, http://glaros.dtc.umn.edu/gkhome/node/85.

[278] K. Schloegel, G. Karypis, and V. Kumar, *Wavefront diffusion and lmsr: Algorithms for dynamic repartitioning of adaptive meshes*, IEEE Transactions on Parallel and Distributed Systems, 12 (2001), pp. 451–466, https://doi.org/10.1109/71.926167.

[279] K. Schloegel, G. Karypis, and V. Kumar, *Graph partitioning for high-performance scientific simulations*, in Sourcebook of Parallel Computing, Morgan Kaufmann Publishers, 2003, ch. 18, pp. 491–541, http://glaros.dtc.umn.edu/gkhome/fetch/papers/gpchapter.pdf.

[280] F. Schornbaum and U. Rüde, *Massively parallel algorithms for the lattice boltzmann method on nonuniform grids*, SIAM Journal on Scientific Computing, 38 (2016), pp. C96–C126, https://doi.org/10.1137/15M1035240.

[281] F. Schornbaum and U. Rüde, *Extreme-scale block-structured adaptive mesh refinement*, SIAM Journal on Scientific Computing, 40 (2018), pp. C358–C387, https://doi.org/10.1137/17M1128411.

[282] C. Schulz, *High quality graph partitioning*, PhD thesis, Karlsruher Institut für Technologie, 2013, http://algo2.iti.kit.edu/schulz/dissertation_christian_schulz.pdf.

[283] *SeiSol: High resolution simulation of seismic wave propagation in realisitic media with complex geometry*, 2021, http://www.seissol.org/ (accessed 2021-10-28).

[284] J. P. Shen and M. H. Lipasti, *Modern processor design: fundamentals of superscalar processors*, Waveland Press, 2013. ISBN 978-1478607830.

[285] M. Shephard, J. Flaherty, H. De Cougny, C. Özturan, C. Bottasso, and M. Beall, *Parallel automated adaptive procedures for unstructured meshes*, Parallel Computing in CFD, 807 (1995), pp. 6.1–6,49, https://scorec.rpi.edu/REPORTS/1995-11.pdf.

[286] M. S. Shephard and M. K. Georges, *Automatic three-dimensional mesh generation by the finite octree technique*, International Journal for Numerical methods in engineering, 32 (1991), pp. 709–749, https://doi.org/10.1002/nme.1620320406.

[287] Z. Shojaaee, M. R. Shaebani, L. Brendel, J. Török, and D. E. Wolf, *An adaptive hierarchical domain decomposition method for parallel contact dynamics simulations of granular materials*, Journal of Computational Physics, 231 (2012), pp. 612–628, https://doi.org/10.1016/j.jcp.2011.09.024.

[288] W. Sierpínski, *Sur une nouvelle courbe continue qui remplit toute une aire plane*, Bulletin International de l'Académie Sciences Cracovie, (1912), pp. 462–478.

[289] H. D. Simon, *Partitioning of unstructured problems for parallel processing*, Computing systems in engineering, 2 (1991), pp. 135–148, https://doi.org/10.1016/0956-0521(91)90014-V.

[290] B. Smith, P. Bjørstad, and W. Gropp, *Domain Decomposition, Parallel Multilevel Methods for Elliptic Partial Differential Equations*, Cambridge University Press, 1996. ISBN 0-521-49589-X, https://www.mcs.anl.gov/~bsmith/ddbook.html.

[291] A. Sohn and H. Simon, *S-harp: A scalable parallel dynamic partitioner for adaptive mesh-based computations*, in SC'98: Proceedings of the 1998 ACM/IEEE Conference on Supercomputing, IEEE Computer Society, 1998, pp. 30–30, https://doi.org/10.1109/SC.1998.10023.

[292] S. Srinivasan, I. Ashok, H. Jônsson, G. Kalonji, and J. Zahorjan, *Dynamic-domain-decomposition parallel molecular dynamics*, Computer physics communications, 102 (1997), pp. 44–58, https://doi.org/10.1006/jpdc.1997.1408.

[293] J. Steensland, S. Chandra, and M. Parashar, *An application-centric characterization of domain-based sfc partitioners for parallel samr*, IEEE Transactions on Parallel and Distributed Systems, 13 (2002), pp. 1275–1289, https://doi.org/10.1109/TPDS.2002.1158265.

[294] H. S. Stone, *High-performance computer architecture*, Pearson Education, 1987. ISBN 9780201526882.

[295] G. O. Strawn, *Grand challenges: Science, engineering, and societal advances requiring networking and information technology research and development*, National Coordination Office for Networking and Information Technology Reseach and Development, (2006), https://www.nitrd.gov/pubs/200311_grand_challenges.pdf.

[296] H. Sundar, R. S. Sampath, and G. Biros, *Bottom-up construction and 2:1 balance refinement of linear octrees in parallel*, SIAM Journal on Scientific Computing, 30 (2008), pp. 2675–2708, https://doi.org/10.1137/070681727.

[297] H. Sutter, *The free lunch is over: A fundamental turn toward concurrency in software*, Dobbs's Journal, (2005), http://www.gotw.ca/publications/concurrency-ddj.htm (accessed 2021-08-31).

[298] V. E. Taylor and B. Nour-Omid, *A study of the factorization fill-in for a parallel implementation of the finite element method*, International journal for numerical methods in engineering, 37 (1994), pp. 3809–3823.

[299] J. D. Teresco, K. D. Devine, and J. E. Flaherty, *Partitioning and dynamic load balancing for the numerical solution of partial differential equations*, in Numerical solution of partial differential equations on parallel computers, Springer, 2006, pp. 55–88, https://doi.org/10.1007/3-540-31619-1_2.

[300] J. D. Teresco and L. P. Ungar, *A comparison of zoltan dynamic load balancers for adaptive computation*, tech. report, Department of Computer Science, Williams College, 2003, https://j.teresco.org/research/publications/lbcompare02/lbcompare02.pdf.

[301] *Top 10 supercomputing sites for june 1993*, 2021, https://www.top500.org/lists/top500/2021/06/ (accessed 2021-07-16).

[302] *Top 10 supercomputing sites for november 1993*, 1993, https://www.top500.org/lists/top500/1993/11/ (accessed 2021-07-16).

[303] A. Trifunović and W. J. Knottenbelt, *Towards a parallel disk-based algorithm for multilevel k-way hypergraph partitioning*, in 18th International Parallel and Distributed Processing Symposium, 2004. Proceedings., IEEE Computer Society, 2004, p. 236, https://doi.org/10.1109/IPDPS.2004.1303286.

[304] A. Trifunović and W. J. Knottenbelt, *Parallel multilevel algorithms for hypergraph partitioning*, Journal of Parallel and Distributed Computing, 68 (2008), pp. 563–581, https://doi.org/10.1016/j.jpdc.2007.11.002.

[305] U. Trottenberg, C. W. Oosterlee, and A. Schuller, *Multigrid*, Elsevier, 2000. ISBN 9780127010700.

[306] Y.-H. Tseng and J. H. Ferziger, *A ghost-cell immersed boundary method for flow in complex geometry*, Journal of Computational Physics, 192 (2003), pp. 593–623, https://doi.org/10.1016/j.jcp.2003.07.024.

[307] *Unified european applications benchmark suite*, 2021, https://repository.prace-ri.eu/git/UEABS/ueabs/ (accessed 2021-11-15).

[308] C. Uphoff, S. Rettenberger, M. Bader, E. H. Madden, T. Ulrich, S. Wollherr, and A.-A. Gabriel, *Extreme scale multi-physics simulations of the tsunamigenic 2004 sumatra megathrust earthquake*, in SC '17: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, Association for Computing Machinery, 2017, pp. 1–16, https://doi.org/10.1145/3126908.3126948.

[309] R. Van Driessche and D. Roose, *Dynamic load balancing with a spectral bisection algorithm for the constrained graph partitioning problem*, in International Conference on High-Performance Computing and Networking, Springer, 1995, pp. 392–397, https://doi.org/10.1016/0167-8191(94)00059-J.

[310] D. Vanderstraeten, C. Farhat, P. Chen, R. Keunings, and O. Ozone, *A retrofit based methodology for the fast generation and optimization of large-scale mesh partitions: beyond the minimum interface size criterion*, Computer methods in applied mechanics and engineering, 133 (1996), pp. 25–45, https://doi.org/10.1016/0045-7825(96)01024-9.

[311] B. Vastenhouw and R. H. Bisseling, *A two-dimensional data distribution method for parallel sparse matrix-vector multiplication*, SIAM review, 47 (2005), pp. 67–95, https://doi.org/10.1137/S0036144502409019.

[312] M. Vazquez, G. Houzeaux, S. Koric, A. Artigues, J. Aguado-Sierra, R. Aris, D. Mira, H. Calmet, F. Cucchietti, H. Owen, A. Taha, and J. M. Cela, *Alya: Towards exascale for engineering simulation codes*, 2014, https://arxiv.org/abs/1404.4881.

[313] R. Verfürth, *A posteriori error estimation and adaptive mesh-refinement techniques*, Journal of Computational and Applied Mathematics, 50 (1994), pp. 67–83, https://doi.org/10.1016/0377-0427(94)90290-9.

[314] D. E. D. Vinkemeier and S. Hougardy, *A linear-time approximation algorithm for weighted matchings in graphs*, ACM Transactions on Algorithms (TALG), 1 (2005), pp. 107–122, https://doi.org/10.1145/1077464.1077472.

[315] C. Walshaw, *Multilevel refinement for combinatorial optimisation problems*, Annals of Operations Research, 131 (2004), pp. 325–372, https://doi.org/10.1007/978-3-540-78295-7_9.

[316] C. Walshaw and M. Cross, *Dynamic mesh partitioning and load-balancing for parallel computational mechanics codes*, Parallel and Distributed Computing for Computational Mechanics, (1999), https://chriswalshaw.co.uk/papers/fulltext/WalshawECMPld99.pdf.

[317] C. Walshaw and M. Cross, *Mesh partitioning: a multilevel balancing and refinement algorithm*, SIAM Journal on Scientific Computing, 22 (2000), pp. 63–80, https://doi.org/10.1137/S1064827598337373.

[318] C. Walshaw, M. Cross, and M. G. Everett, *Parallel dynamic graph partitioning for adaptive unstructured meshes*, Journal of Parallel and Distributed Computing, 47 (1997), pp. 102–108, https://doi.org/10.1006/jpdc.1997.1407.

[319] C. Walshaw et al., *Parallel jostle userguide*, University of Greenwich, London, (1998).

[320] C. H. Walshaw, M. Cross, and M. G. Everett, *A localized algorithm for optimizing unstructured mesh partitions*, The International Journal of Supercomputer Applications and High Performance Computing, 9 (1995), pp. 280–295, https://doi.org/10.1177/109434209500900403.

[321] M. S. Warren and J. K. Salmon, *A parallel hashed oct-tree n-body algorithm*, in Proceedings of the 1993 ACM/IEEE conference on Supercomputing, Association for Computing Machinery, 1993, pp. 12–21, https://doi.org/10.1145/169627.169640.

[322] J. Watts, M. Rieffel, and S. Taylor, *A load balancing technique for multiphase computations*, Proceedings High Performance Computing '97, (1997), pp. 15–20, https://surface.syr.edu/lcsmith_other/18.

[323] J. Watts and S. Taylor, *A practical approach to dynamic load balancing*, IEEE Transactions on Parallel and Distributed Systems, 9 (1998), pp. 235–248, https://doi.org/10.1109/71.674316.

[324] T. Weinzierl, *The peano software: Parallel, automaton-based, dynamically adaptive grid traversals*, ACM Transactions on Mathematical Software, 45 (2019), pp. 14:1–14:41, https://doi.org/10.1145/3319797.

[325] T. Weinzierl and M. Mehl, *Peano – A Traversal and Storage Scheme for Octree-Like Adaptive Cartesian Multiscale Grids*, SIAM Journal on Scientific Computing, 33 (2011), pp. 2732–2760, https://doi.org/10.1137/100799071.

[326] P. Wenisch, *Computational steering of CFD simulations on teraflop-supercomputers*, PhD thesis, Technische Universität München, 2008.

[327] S. R. Wheat, *A fine-grained data migration approach to application load balancing on MP MIMD machines*, PhD thesis, The University of New Mexico, 1992.

[328] S. R. Wheat, K. D. Devine, and A. B. Maccabe, *Experience with automatic, dynamic load balancing and adaptive finite element computation*, tech. report, Sandia National Labs, 1993, https://www.osti.gov/biblio/10102903-experience-automatic-dynamic-load-balancing-adaptive-finite-element-computation.

[329] M. H. Willebeek-LeMair and A. P. Reeves, *Strategies for dynamic load balancing on highly parallel computers*, IEEE Transactions on parallel and distributed systems, 4 (1993), pp. 979–993, https://doi.org/10.1109/71.243526.

[330] M. Wolf, *Computers as components: principles of embedded computing system design*, Elsevier, 2012. ISBN 978-8181475794.

[331] J. Wolfson-Pou, *Asynchronous versions of Jacobi, multigrid, and Chebyshev solvers*, PhD thesis, Georgia Institute of Technology, 2020.

[332] J. Wolfson-Pou and E. Chow, *Convergence models and surprising results for the asynchronous jacobi method*, in 2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS), IEEE Computer Society, 2018, pp. 940–949, https://doi.org/10.1109/IPDPS.2018.00103.

[333] J. Wolfson-Pou and E. Chow, *Asynchronous multigrid methods*, in 2019 IEEE international parallel and distributed processing symposium (IPDPS), IEEE Computer Society, 2019, pp. 101–110, https://doi.org/10.1109/IPDPS.2019.00021.

[334] J. Wolfson-Pou and E. Chow, *Modeling the asynchronous jacobi method without communication delays*, Journal of Parallel and Distributed Computing, 128 (2019), pp. 84–98, https://doi.org/10.1016/j.jpdc.2019.02.002.

[335] C. Xu and F. C. Lau, *Load balancing in parallel computers: theory and practice*, vol. 381, Springer, 1996, https://doi.org/10.1007/b102252.

[336] C. Xu, F. C. Lau, and R. Diekmann, *Decentralized remapping of data parallel applications in distributed memory multiprocessors*, Concurrency: Practice and Experience, 9 (1997), pp. 1351–1376. https://doi.org/10.1002/(SICI)1096-9128(199712)9:12<1351::AID-CPE283>3.0.CO;2-C.

[337] F. Xu, D. Schillinger, D. Kamensky, V. Varduhn, C. Wang, and M.-C. Hsu, *The tetrahedral finite cell method for fluids: Immersogeometric analysis of turbulent flow around complex geometries*, Computers & Fluids, 141 (2016), pp. 135–154, https://doi.org/10.1016/j.compfluid.2015.08.027.

[338] I. Yamazaki, E. Chow, A. Bouteiller, and J. Dongarra, *Performance of asynchronous optimized schwarz with one-sided communication*, Parallel Computing, 86 (2019), pp. 66–81, https://doi.org/10.1016/j.parco.2019.05.004.

[339] D. Yeung, W. J. Dally, and A. Agarwal, *How to choose the grain size of a parallel computer*, tech. report, MIT Lab oratory for Computer Science and Articial Intelligence Laboratory, 1994, https://user.eng.umd.edu/~yeung/papers/grain.pdf.

[340] N. Zander, *Multi-level hp-FEM: dynamically changing high-order mesh refinement with arbitrary hanging nodes*, PhD thesis, Technische Universität München, 2016, https://mediatum.ub.tum.de/1335288.

[341] N. Zander, T. Bog, M. Elhaddad, F. Frischmann, S. Kollmannsberger, and E. Rank, *The multi-level hp-method for three-dimensional problems: Dynamically changing high-order mesh refinement with arbitrary hanging nodes*, Computer Methods in Applied Mechanics and Engineering, 310 (2016), pp. 252–277, https://doi.org/10.1016/j.cma.2016.07.007.