Diese Arbeit wurde vorgelegt am
Lehrstuhl für Hochleistungsrechnen (Informatik 12), IT Center.

Usecase Analyse und Hyperparameter Optimierung von targetDART Anwendungen

Usecase Analysis and Hyperparameter Optimization for targetDART Applications

Bachelorarbeit

Felix Knierim
Matrikelnummer: 445677

Aachen, den 13. November 2025

Erstgutachter:	Prof. Dr. rer. nat. Matthias S. Müller (')
Zweitgutachter:	Prof. Michael Bader
Betreuer:	Adrian Schmitz, M.Sc. (')

(') Lehrstuhl für Hochleistungsrechnen, RWTH Aachen University
IT Center, RWTH Aachen University
(*) Lehrstuhl für Betriebssysteme, RWTH Aachen University
communicated by Prof. Matthias S. Müller

Ich versichere hiermit, dass ich die vorliegende Arbeit selbständig und ohne Benutzung anderer als der angegebenen Hilfsmittel angefertigt habe. Alle Stellen, die wörtlich oder sinngemäß aus veröffentlichten und nicht veröffentlichten Schriften entnommen sind, sind als solche kenntlich gemacht. Die Arbeit ist in gleicher oder ähnlicher Form noch nicht als Prüfungsarbeit eingereicht worden.

Aachen, den 13. November 2025

Abstract

For all kinds of projects that require a lot of computations, multiple computation devices are used in parallel, in order to speed up the execution. Not just multiple nodes, but also accelerators and CPUs are used simultaneously. Load imbalances often occur on applications that utilize multiple devices. The occurrence of load imbalances means that a computation unit has more workload than another computation unit. Load imbalances slow down the executions because computing devices, which have less workload than other devices, will finish earlier and stay idle. If a device stays idle while others are processing workload, the runtime can be decreased by offloading parts of the workload to the idle devices. A common cause for load imbalances is that they are induced by the algorithm that the application uses or by external factors, like a computation device that processes computations faster or slower than others.

The workload can be balanced during the execution of the program, which is called dynamic load balancing. Static load balancing balances the workload before it is executed and cannot react to imbalances that occur during the execution. Most of the tools that apply dynamic load balancing balance the load between CPU and an accelerator or between multiple nodes. A solution that is able to apply both is targetDART. targetDART provides runtime support that applies dynamic load balancing within a node, between CPU and accelerators, and across different nodes. It is integrated into the LLVM project and is available as a plugin for the clang compiler. This thesis conducts benchmarks to test the capabilities and limits of targetDART regarding different use cases. Furthermore, the hyperparameters of targetDART are also tested, in order to provide guidelines for their usage.

The results of this thesis show that targetDART scales efficiently. Although a lot of factors that depend on the use case induce small overheads of 14-16% or 400ms. The use case that induced high overhead was for using CPU-only as a computation device. Another plugin that targetDART utilizes induces that overhead. For the hyperparameters, a comparison of a naive case with no hyperparameters set and a best case, which combines all optimal hyperparameter settings, shows that with the guidelines for hyperparameters, a speedup of 2.06× can be achieved.

Keywords: HPC, OpenMP, MPI, targetDART, load balancing, dynamic load balancing, device offloading

Contents

List of Figures

List of Tables

1. Introduction

In order to execute computations faster, programs are parallelized. Throughout the past, this approach has proved itself to be more scalable and energy efficient. Programs are parallelized on multiple cores of a single CPU, but different computation devices like GPUs are also used to speed up the execution. If a different computation device is used, it can run in parallel with the CPU on different computations. Another approach to parallelize the computations is to use multiple computers. This way, the computations are executed on multiple CPUs. If multiple computation devices are used (which also includes multiple CPUs), so-called load imbalances can occur. A load imbalance occurs if a computation device has more computations to process than another device. Load imbalances extend the execution time, since some devices have more workload than others, which will finish their computations earlier. Devices with a lot of workload take more time, while other devices remain idle. The computation can be accelerated by balancing the workload. These load imbalances can occur during the program execution between different computers. A common cause of load imbalances is that the duration of the computations depends on the input data. Since every node is dedicated to a fixed part of the input data, some nodes get more workload than others. The parts of the input that cause more computation are often unknown before the computation begins, making it impossible to balance the workload beforehand.

An example of such computation methods is adaptive mesh refinement [7]. It refines the accuracy of the result values of some subspaces of a mesh. Which parts of the mesh are refined are chosen by the program based on the previous results and, therefore, it is not possible to predict which parts will cause more workload than others, until prior computations finish. External causes for load imbalances, like power capping, which is sometimes applied in datacenters, can also occur during execution.

To balance the workload during execution, dynamic load balancing has to be applied. Dynamic load balancing balances the workload between multiple computing devices during the computations. Static load balancing, on the other hand, can only balance the workload before the computations start. Besides balancing the workload between different nodes, it is also possible to balance the workload between different computation devices, such as CPUs and GPUs. targetDART [25], which is developed by the RWTH Aachen's chair for High Performance Computing, can dynamically load balance between different computation devices and between different nodes. Furthermore, it is integrated into the LLVM project as a plugin and utilizes OpenMP pragmas. With the OpenMP pragmas, it is easy to use targetDART and it generates low overhead because of the integration into the compiler.

In the work of Schmitz et al. [25], targetDART was tested with different benchmarks and with a case study using ExaHyPE [23]. This thesis aims to conduct more benchmarks in order to provide a better understanding of targetDART's behavior. With the results of the benchmarks, suitable use cases for targetDART are determined. Furthermore, the benchmarks test the impact of various hyperparameters of targetDART in order to tune the execution time. With the results of both, guidelines for the use cases and the optimal hyperparameter tuning are derived.

The following chapter 2 provides an overview of existing solutions for dynamic load balancing and prior work to targetDART. After that, the background section 3 explains the technical basics that are needed to understand targetDART. The background section also describes how targetDART works and focuses on runtime support, task management and hyperparameters. At the end of the Background section, the benchmarks and their results from previous work are presented. In the benchmark section 4, which follows after that, the setup is described, as well as every benchmark that was conducted. The description of the setup first describes a base scenario, which is the base for every other benchmark, with single parameters of the setup being derived from that base scenario. In the benchmark section, the results are presented and discussed. The guidelines that are derived for the use cases and hyperparameters are presented in that section as well. At the end, a final conclusion in Chapter 5 summarizes the key findings and lists possibilities for further research.

2. Related Works

In the work of Schmitz et al. [25], the author shows that there are many other tools that have introduced the tasking concept and allowed the offloading of these to an accelerator, like a GPU or another node. One example is HPX, which enables task offloading over MPI [1]. HPX can also be combined with OP2 [21], a framework that translates written code into different parallel implementations, which can also utilize GPUs. With the combination of these tools, it is possible to migrate tasks to MPI nodes, which can then utilize GPUs to compute the tasks [15]. To evaluate this approach, the authors used the Airfoil application to measure the performance gain with strong and weak scaling. But no systematic testing of the possible use cases of this tool was conducted, nor of the hyperparameters that HPX and OP2 have. HPX itself has a few more case studies that compared HPX mostly with handwritten OpenMP or OpenMP+MPI versions [16]. Kalkhof et al. [14] considered in their case study with HPX various accelerators like GPUs, FPGAs and AI engines, whereby most case studies are limited to using only GPUs as accelerators since they are the most common ones in datacenters.

The work of Samfass et al. [24] focused on load balancing between MPI nodes without considering GPU offloading. They proposed multiple load balancing algorithms, which are refined versions of existing algorithms or propose additions like blacklisting to prevent overbooking of single nodes. In the paper, they evaluate their approach with a simulation in ExaHyPE.

DCUDA [30] is a wrapper library for CUDA that performs dynamic load balancing between multiple GPUs. The tool is also able to load balance between multiple applications, but unlike the previous example, it is not able to load balance between multiple nodes. The performance of the tool was compared to the static load balancing algorithm, least-loaded, but was also tested with regard to the overhead of the library and the fairness between different applications. About 20 benchmarks from the CUDA samples were taken for the measurements, so DCUDA was tested with different kinds of small real-world use cases.

The tool StarPU [27] focuses on interdisciplinary use cases. It was designed to process real-time data of autonomous driving vehicles. It was implemented as a library for task scheduling on heterogeneous systems to provide fast data processing and decision-making for autonomous vehicles. The tool was tested on a simulator that provides realistic scenarios for autonomous vehicles.

Another example of interdisciplinary tools that aims to load balance tasks is the work of Thavappiragasam et al. [28]. In this paper, they propose a load balancing library for applications in bioinformatics, like sequence analysis and protein structure prediction. The tool was implemented using OpenMP and focuses on dynamic

load balancing between multiple GPUs on a single node. Variable task sizes and convergence rates are common for applications in bioinformatics, so the tool was also tested with matrix multiplications, which have variable input sizes and a random number of matrices that are multiplied to simulate a convergence rate.

The tool IRIS [17] has a similar range of features compared to targetDART, which includes static and dynamic load balancing across multiple nodes and accelerators. It focuses on supporting as many devices with different runtime libraries as possible. It supports, for example, OpenMP, CUDA, HIP, OpenCL and SYCL but also more devices. The tool is implemented as a framework and does not require source code modification for integration, but it also supports complex execution chains. In the work of Jungwon et al. [17], they conducted benchmarks which measured the overhead compared to handwritten code and the potential speedup that is achievable by load balancing. They also conducted more benchmarks in a different paper [13] where they compared the peak performance for a simple use case with more complex task chains to see how the tool performs with increased complexity. In that paper, the performance of different scheduling strategies was also compared. IRIS was also integrated in some toolchains [9], [8], [22] and was used in a case study [19] with ProtoX [20] to compute 3-dimensional Euler equations.

Just like the previous examples, most of the tools that try to load balance between nodes or GPUs have a case study and a few tests, but no systematic testing of changes in the environment and the workload. This also holds for targetDART. In the work of Schmitz et al. [25], several benchmarks were performed that aim to prove that targetDART is suitable for load balancing across multiple nodes with accelerators. It also has a case study: Simulating the Tohoku tsunami in ExaHyPE. But the paper does not systematically test for which use cases the tool is suited or which hyperparameter might increase the performance.

Hyperparameter optimization is also important for other fields of computer science, like machine learning. There are some strategies in machine learning for searching through the hyperparameter space, like grid search, where all possible combinations of parameters are systematically tested, or random search, where randomized configurations of hyperparameters are tested. Since a full grid search is often too much computational effort, the configurations are often manually chosen. For random search, it is surprisingly efficient compared to grid search [11] because often just a few hyperparameters have a high impact on the precision of a model. Random search explores the dimensions of these few important parameters with less computation effort than grid search. Although grid search and random search have not been used in this thesis, it shows that exploring the entire hyperparameter space is often unnecessary.

This thesis will not conduct any further case studies and instead use simple test programs to limit the complexity of the benchmarks. It aims to supplement the tests of the previous work of Schmitz by further exploring the performance under various circumstances. The effect of hyperparameters is also tested to determine in which ways they can influence the performance. With that insight, guidelines on how to use targetDART in the best way possible can be provided.

3. targetDART and the Underlying Hardware Concepts

This chapter will explain the fundamentals of memory architecture that are needed to understand the performance behaviour and it will also explain targetDART itself. First, the memory topology that is relevant for the benchmarks is explained. This includes the difference between virtual and physical memory, as well as page locked memory, but also the NUMA architecture. Then the following section will explain how targetDART works in general, but will also go into detail for the runtime support and the load balancing. The explanation of targetDART will also introduce the hyperparameters of targetDART and present the benchmarks of the work of Schmitz et al. [25].

3.1. Memory

This section will explain the difference between virtual and physical memory and the implications this has for performance. Furthermore, it will explain how the NUMA architecture works and its influence on performance.

3.1.1. Virtual and Physical Memory

Physical memory refers to all kinds of hardware memory that are built into a computer, for example, the RAM or the GPU memory. Therefore, the physical memory addresses directly point to a location in the hardware memory.

Virtual memory refers to an address space that is created by the OS. This address space can be larger than the space of physical addresses, so programs that only use the virtual address space can allocate more memory than is physically available. In that case, the data can also be stored on a hard drive (called swap memory). Virtual memory addresses point to the virtual address space and need to be translated by the CPU to get the physical address and the data that is stored there [29]. The data, to which the virtual addresses point, may be moved by the OS during runtime.

When offloading data to a GPU, it can be CPU-driven or GPU-driven offloading. CPU-driven offloading utilizes a virtual address, and the CPU translates it as usual. GPU-driven offloading is done by the GPU, which means that the addresses are not translated by the CPU and the GPU copies the data directly. During the copy process, it is not allowed to move the data, like the OS sometimes does, so the physical address needs to stay the same. Pinned virtual memory fulfills these

requirements by locking the position of the data in the physical memory. If memory is unpinned, which is the default, then the memory first needs to be copied to a pinned memory address before it is offloaded. With pinned memory, a GPU can also access the data without a translation of the CPU and can copy the data asynchronously.

3.1.2. NUMA nodes

The concept of NUMA (Non-Uniform Memory Access) nodes was invented to address the problem of the memory wall [2]. This problem is caused if multiple CPU cores, which can work independently, need to share the hardware bus to the memory and therefore the bandwidth.

The solution to this is to partition the cores of a CPU and the memory into NUMA nodes and give each NUMA node its own link to its memory section, which leads to a higher bandwidth per core. NUMA nodes also have interconnects to memory sections of other NUMA nodes, but this link is slower and has a lower bandwidth. This architecture is called NUMA[12].

If a node has multiple CPUs and each of them has NUMA nodes, the bandwidth between NUMA nodes of different CPUs is lower than the interconnect between NUMA nodes on the same CPU. For a single device that is connected by PCIe, there are not multiple PCIe buses to each NUMA node; instead, the device has a single link to one of the NUMA nodes. So if a process utilizes this device, the latency and bandwidth are better if the process runs on the NUMA node to which the device is connected. [12]

3.2. TargetDART

targetDART [3] is implemented as an LLVM plugin and utilizes OpenMP pragmas for offloading to a device and MPI for communication. The LLVM project is a library that is used for building compilers, such as the clang compiler. The goal of targetDART is to dynamically balance a given workload by offloading tasks to GPUs, as well as to other MPI nodes. It accomplishes that by providing runtime support that handles the load balancing and execution of tasks.

An example of a program that utilizes targetDART for matrix multiplications is in Listing 3.1. As illustrated in the example, the setup for MPI must be implemented in the program with the initialization and the finalization at the end. Just like for OpenMP, the code is executed on a single core until an OpenMP pragma is used. In the example, the code before line 19 and after line 27 is executed on multiple MPI nodes, but on a single core. In line 19, the pragma is used to generate a single task that will execute the following for loop in parallel. In total, the pragma is executed 10 times, so 10 tasks are generated. The device to which these tasks are offloaded is named `TARGETDART_ANY` and refers to a targetDART device, which, in this case,

executes the tasks on CPUs and GPUs. targetDART offers several targetDART devices to which a user can offload tasks:

- `TARGETDART_CPU`: The tasks will be executed only on the CPU.
- `TARGETDART_OFFLOAD`: The tasks will only be executed on accelerators like GPUs.
- `TARGETDART_ANY`: The tasks will be distributed to CPUs and accelerators
- `TARGETDART_DEVICE(n)`: The tasks will be executed on the device with the id `n`. It works similarly to the OpenMP clause `device(n)`, but the tasks will be considered at the load balancing process. The tasks will not be migrated to a different node.
- `TARGETDART_X_LOCAL`: The X can be replaced with one of the first three options of this list, for example `TARGETDART_CPU_LOCAL`. The tasks will be executed on the corresponding device(s), but they will not be migrated to a different node.

The remaining part of the pragma is designed identically to GPU offloading. The kernel of the matrix multiplication does not need to be modified.

The following subsection will explain the task queues of the targetDART devices and their priorities, as well as the threads that organize the execution and scheduling of tasks. For this explanation, it will take Listing 3.1 as an example and provide a rundown of the most significant events. Next, the two scheduling algorithms that are used by targetDART are explained. At last, the hyperparameters of targetDART are explained and the benchmarks and the case study that were presented in the work of Schmitz et al. [25] are explained briefly.

3.2.1. Runtime Support

This section will describe the runtime support in detail. It will focus on the runtime threads that handle the execution and load balancing, the task queues and the task structure. To illustrate how these parts work together, an example run of a process that utilizes targetDART will illustrate the purpose of all components.

Runtime Threads

The targetDART runtime support operates three different types of runtime threads. The scheduler thread coordinates the dynamic load balancing, together with the receiver thread. The executer threads are responsible for dispatching the execution of the code on the devices. All threads and their relation to each other are depicted in Figure 3.1. In that figure, it is illustrated that the executor threads take tasks from the task queues of their devices and use another plugin to dispatch the execution of the task. For each distinct physical device, there is at least one executer thread,

Listing 3.1: An example of C++ matrix multiplication which utilizes targetDART - copied from [25]

```
#include <stdlib.h>
#include <cstdlib>
#include <omp.h>
#include <mpi.h>

int main(int argc, char** argv) {
    int rank, size;
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD , &rank);
    MPI_Comm_size(MPI_COMM_WORLD , &size);
    int d = 1000;
    int ntasks = 10 * rank;
    double *A =(double *) malloc(d * d * sizeof(
       double));
    double *B =(double *) malloc(d * d * sizeof(
       double));
    double *C =(double *) malloc(ntasks * d * d *
       sizeof(double));

    for(int t = 0; t < ntasks; t++) {
    double *C_l = C + t * d * d;
        #pragma omp target teams distribute parallel for
        map(from:C_l[0:d*d]) map(from:C_l[0:d*d])
        map(to:A[0:d*d]) map(to:B[0:d*d]) map(to:d)
        device(TARGETDART_ANY) collapse(2) nowait
        for(int i = 0; i < d; i++) {
            for(int j = 0; j < d; j++) {
                C_l[i * d + j] = 0;
                for(int k = 0; k < d; k++) {
                    C_l[i * d + j] += A[i * d + k] *
                       B[k * d +j];
                }
            }
        }
    }

    #pragma omp taskwait
    MPI_Barrier(MPI_COMM_WORLD);
    free(A); free(B); free(C);
    MPI_Finalize();
    return 0;
}
```

although not all of them are displayed in the figure. For accelerators, multiple threads can be used; for CPUs, just one thread per node can be used. After the tasks are executed, the results are returned. If a task was migrated, the executer thread sends the results to the receiver thread of the node on which the task was created.

The receiver thread receives not just results from offloaded tasks, but it also receives tasks that are migrated to a node. The scheduler thread runs the load balancing algorithm. If the algorithm decides to migrate tasks, the scheduler thread takes a task from a task queue and sends it to the receiver thread of the other node. The receiver thread sorts the received tasks into the task queues. This process of a task migration is illustrated in the figure for two nodes, but the task queues for the second node are left out. For the second node, all works identically to the first node.

Task Queues

The targetDART runtime is supported by the plugin manager of the LLVM project. The plugin manager will generate the tasks from the OpenMP pragmas and forward them to targetDART. The tasks are sorted into the task queues of the corresponding devices. This is also illustrated in Figure 3.1, but unlike the figure suggests, there are more than three task queues. For each device that can be accessed with the `TARGETDART_DEVICE(n)` device, there is a task queue. For the targetDART devices `TARGETDART_CPU`, `TARGETDART_OFFLOAD` and `TARGETDART_ANY`, there are three task queues each: local, remote and migratable. The "local" task queues are for tasks that were created with the `TARGETDART_X_LOCAL` device. The remote queues are for tasks that were generated on a different node and were migrated to this node. The migratable queues are for the remaining tasks. The task queues also have different priorities that determine which tasks are executed first. [3]

Task Structure

When the program is executed and reaches the OpenMP pragmas, the plugin manager of the LLVM project will generate the tasks. All tasks consist of six components:

- A uid that identifies a task. The uid consists of the MPI rank of the node, where the task was created and an identifier that identifies the task on that node.
- A host base pointer that is an address offset for the computation kernel within the binary. To execute the code of the kernel on a different node, the position of the kernel within the binary is stored in the host base pointer. So if the task is migrated, the host base pointer is added to the address of the entry point for the binary on that node and the result will be the address of the kernel.
- The KernelArgs struct, which is defined by liboffload. It contains launch configurations of the kernel, as well as pointers to the input parameters.

- The device affinity that defines on which device the kernel is executable.
- The Loc data, which holds location data for debugging purposes.
- A return code that signals if the kernel was executed successfully or which error occurred.

An Execution Example

To illustrate how the runtime support handles the load balancing and the execution of the tasks, an example where multiple tasks are generated on two nodes that have Nvidia GPUs as accelerators is introduced. For this example, the code of Listing 3.1 is used, but with a small modification: The first rank starts with significantly more tasks than the second rank. The example will focus on one specific task that will be offloaded to the other node and will be executed on a GPU.

The task was generated on node 0 and is identified on that node with the identifier 3, so the uid of that task is 03. Since the tasks were generated with the targetDART device `TARGETDART_ANY`, the plugin manager will put them all into the migratable queue of the `TARGETDART_ANY` device. The tasks also get the affinity `ANY`. In this case, the host base pointer would refer to the code line 20, because the kernel starts there. Since the host base pointer is a relative address, the address 0 would, for this address, refer to the beginning of the binary.

The task queues also have different priorities to determine which tasks will be executed first. The priorities of the task queues are illustrated in Figure 3.2. The task queues of the `TARGETDART_DEVICE(n)` device have the highest priority, followed by the device queues of the `TARGETDART_CPU` and `TARGETDART_OFFLOAD` devices, which have the same priority. The task queues of the `TARGETDART_ANY` device have the lowest priority.

For the last three devices, all of their task queues have the same priority ordering. In the figure, the task queues of the last three devices are drawn within the box of the device, but the box itself is not a task queue. The priority ordering of the queues of the CPU, offload and "any" devices is that the local queue has the highest, the remote queue the second highest and the migratable the lowest priority. [3]

So for this example, all tasks have the lowest priority, because they are in the migratable queue of `TARGETDART_ANY`. At this moment, it does not make a big difference because all tasks are in the same queue.

As described previously, the first rank has significantly more tasks than the second one. So when the scheduling algorithm is executed by the two scheduler threads of both ranks, they will decide that rank 0 needs to migrate some tasks to rank 1. How the scheduling algorithms work is explained in Section 3.2.2.

To migrate the tasks to the other node, the scheduler thread of rank 0 takes the tasks from the migratable queues of the any device and sends them to the receiver thread of rank 1 [25]. This process is drawn in Figure 3.1, but in this figure, the number of task queues is reduced. To send a task, the scheduler thread of rank 0 will first send the sizes of the data in the map clauses and wait for an answer that

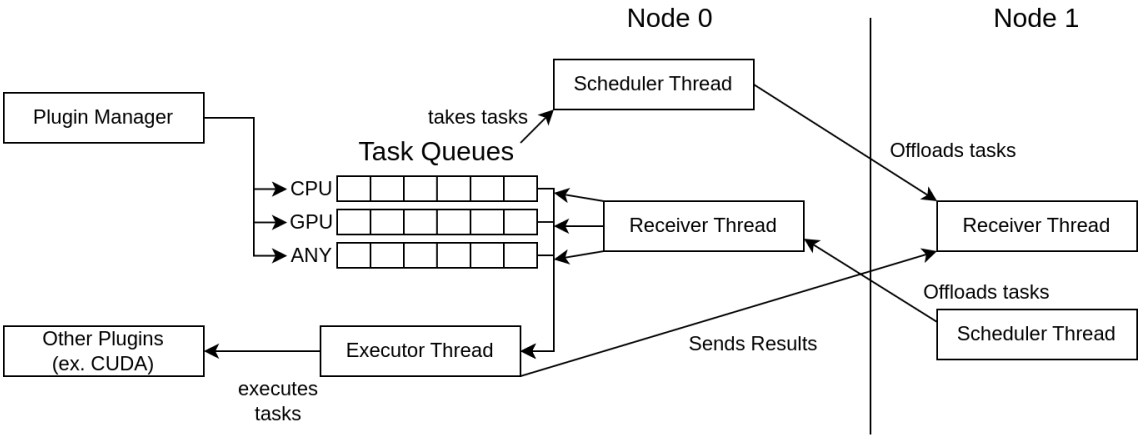


Figure 3.1.: All runtime threads and their relation to each other. For simplification, not all task queues are displayed. Inspired by figure 1 of [25]

will tell the scheduler if there is enough memory available on node 1. After that, the sender will send the task components and all the input data that were declared by the map clauses. The host base pointer of each task is added to the entry address of the binary in order to get the address of the task kernel, so it can be executed. The receiver thread puts the task into the remote queue of the device to which the task was assigned, like `TARGETDART_ANY` in this case.

The task with uid 03 was migrated first and now has a higher priority than the other tasks, which were created on rank 1. So as soon as a device finishes its task, the executer thread of that device will take task 03 from the remote queue before taking one of the migratable queue. Like depicted in the figure, they might also use other plugins to execute a task.

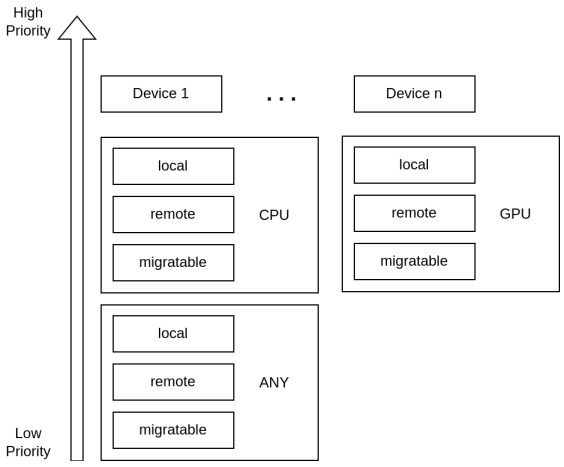


Figure 3.2.: The priorities of the task queues sorted. The figure is self-drawn.

In our example, an Nvidia GPU of node 1 finishes its task before the CPU, so it will take task 03 from the remote queue of the "any" devices. For offloading the task to the GPU, the CUDA plugin is used. The data and the kernel are offloaded to the GPU and the result is computed and sent back to the executer thread. The executer thread will then send the result back to the receiver thread of the first node with the return code for success. When all tasks are finished, the program ends.

3.2.2. Load Balancing

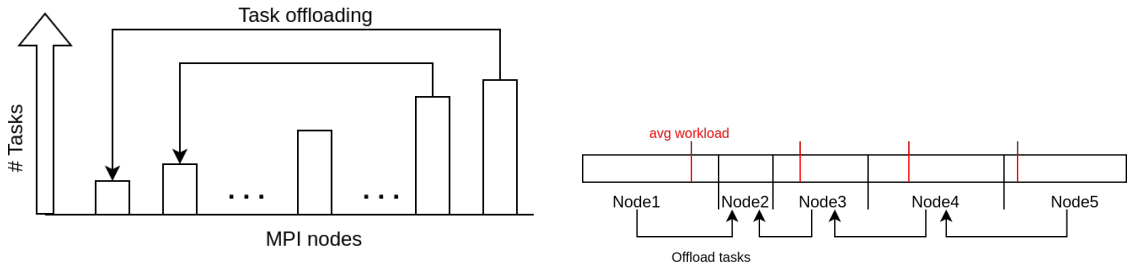
When a program generates tasks that need to be computed on nodes, the tasks need to be distributed to the nodes. There are two ways to distribute the tasks to the nodes. The first one is called static load balancing and it assigns a fixed amount of tasks to each node before the computation begins. This can happen, for example, when coding the mapping in the source code or at runtime, just before the computation starts. The second option is dynamic load balancing, which may reassign tasks during runtime. This is used for adapting to load imbalances that occur during runtime. For example, if some of the tasks require more computation time than others or one node computes the workload more slowly for reasons like power capping, it might happen that one node takes significantly more time than the others. This would slow down the overall computation time, so a dynamic load balancing algorithm could reassign tasks from a node that slows the computation down to nodes that will finish earlier. This way, the node that slows down the computation has less workload and will finish earlier, while the other nodes will have less idle time. A static load balancing algorithm cannot do this after the computation starts and can therefore not handle load imbalances that occur during runtime. But dynamic load balancing also induces more overhead compared to static load balancing, which makes it more suitable for potentially high load imbalances.

targetDART applies load balancing for distributing tasks between CPUs and accelerators, and between different nodes. For balancing between CPU and accelerators, targetDART utilizes task stealing, which means that a device would take tasks if it has nothing to process left. The implementation of task stealing in targetDART was described in Section 3.2.1, where tasks that are assigned to the "any" device are in a queue from which the executer threads from the CPU and accelerators could take them. For load balancing the tasks between computation nodes, targetDART uses task offloading, which means that if a node has too many tasks, it would give tasks to other nodes. targetDART offers two different algorithms for task offloading, which will be presented in the following sections.

Fine Grained Scheduling

This algorithm was defined by Klinkenberg et al. [18]. For the first step of this algorithm, every node needs to get the computational workload of all nodes, which is realized with one of MPI's collectives: `MPI_Allgather()`. The second step is that each node applies a stable sorting algorithm to the list that was acquired from `MPI_Allgather()`. On the last step, each node selects a "victim" for task migration, so the node to which it will migrate tasks. These three steps are executed permanently by every scheduler thread.

For the last step, the upper half of the sorted list selects a victim by taking the highest index of the list and subtracting their own index from it (assuming the list index starts at 0). In Figure 3.3a, the resulting mapping is illustrated. Each node that migrates tasks to another node will migrate just one task per iteration. But it



(a) A figure of the task migration phase of the fine grained load balancing, where each node decides where it will migrate tasks to.

(b) A figure of the task migration phase of the coarse grained load balancing after the prefix sum and the average workload were computed.

Figure 3.3.: Both load balancing algorithms in the migration phase. Both inspired by Figure 2 of [25]

also causes overhead to migrate tasks, so for a small number of tasks, it would not benefit the performance to migrate them. Therefore, a threshold is introduced. The difference in workload of a node and its migration victim must exceed this threshold to trigger the task migration. For targetDART, this threshold is set to 3 tasks.

The complexity of this algorithm is $O(n \cdot log(n))$ with n being the number of nodes. The complexity is determined by the number of messages that are exchanged, which is $O(log(n))$ and the sorting algorithm, which is $O(n \cdot log(n))$. But it can be reduced to a minimum of $O(log^2(n))$ by implementing parallel sorting [26].

This scheduling algorithm is the default option in targetDART and is deactivated if the environment variable `SKIP_FINE_GRAINED_SCHEDLING` is set to a non-null value.

Coarse Grained Scheduling

The coarse grained scheduling algorithm as defined by Harlacher et al. [10] is based on space-filling curves (SFC). The total workload is represented as a one-dimensional SFC, which is used to determine which process needs to offload tasks to its neighbours.

For the algorithm, the nodes are first aligned in a row and the prefix sum of the workload is computed. The prefix sum computes for a node the sum of all tasks that the previous nodes have combined with its own workload. The formal definition of the prefix sum is $prefix(I) = \Sigma_{i=0}^{N-1} w_i$, whereby I is the node for which the prefix sum is computed and w_i is the workload of the node with index i. So with the prefix sum, each node can compute how much workload all previous nodes have and knows where its own subspace in the SFC starts and ends. The prefix sum is collected with the MPI call `MPI_EXscan()`.

Besides the prefix sum, the total sum of all workloads of every node is collected using `MPI_Allreduce()` with the sum as the reduction operation. With the total

sum, the average workload is computed, which is considered as the target workload. With the average workload as the target, each node can deduce where its own subspace would begin and end if every node had the average workload. In Figure 3.3b, there is an example of how an SFC with multiple nodes looks. The borders of the subspaces of each node are marked in black and the red markers indicate the subspaces that would result if every node had the average workload.

Each node will send tasks to a neighbour if its border exceeds the optimal border for the average workload. In Figure 3.3b, for example, Node1 exceeds the optimal border for its subspace and therefore offloads tasks to Node2, so the borders of the subspace would match the optimal border. This also means that a node can send tasks to another node while receiving tasks from another node, like for Node3, which induces more overhead.

The complexity of the algorithm is mostly determined by the complexity of the two MPI calls, which together have a complexity of $O(log(n))$.

In targetDART, this balancing algorithm is deactivated by default and needs to be called in the source code. If the function for coarse grained scheduling is called, the algorithm is used for the next five iterations and then the iterative scheduling is used as normal (if not deactivated).

3.2.3. Hyperparameters

Hyperparameters are parameters for a process that are not the input for an algorithm, but they configure how the process works. The LLVM project already has a lot of hyperparameters and most of them do not change the way that targetDART works. This section will present the most significant parameters for targetDART, which will be used to tune the performance.

TargetDART inherited some hyperparameters from OpenMP, like the environment variables `OMP_NUM_THREADS` and `OMP_NUM_TEAMS`. The first one defines the maximum number of generated threads when a `parallel` construct is called. The second one defines the maximum number of teams, which are created when using the `teams` construct [4].

Besides the inherited hyperparameters, there are also four more hyperparameters. The first of them directly determines on which CPU cores the runtime threads, scheduler, receiver and executer threads are placed and is called `TD_MANAGEMENT`. This hyperparameter is set as an environment variable, so it does not need to be set while compiling the code, but during execution. The indices of the cores must be listed for this parameter in this order: scheduler thread, receiver thread, all GPU executer threads, CPU executer thread. As an example, let's assume that `TD_MANAGEMENT=5,6,8,9,7` is set on a node with two GPUs. Then the scheduler thread would be pinned to core 5, the receiver thread on core 6 and the CPU executer thread is placed on core 7. The two executor threads for the GPUs are placed on the cores 8 and 9. [25] If this variable is not set, the second hyperparameter determines the placement of the executer threads.

The second hyperparameter is called `TD_EXECUTOR_NPROCS` and determines how

many cores are used for the executor threads, but it will just be used if the hyperparameter TD_MANAGEMENT is not set. If TD_MANAGEMENT is not set, the scheduler thread is placed at core 0, the receiver thread is pinned to core 1 and all of the executer threads are evenly distributed to a number of cores that is defined by TD_EXECUTOR_NPROCS. The first core that would be used by the executer threads is core 2 and the other cores that are used will have an increasing CPU-index from there on. If there are more executer threads than available CPU cores as specified by TD_EXECUTOR_NPROCS, then some executer threads need to share a core and the threads are evenly distributed to the cores. If TD_EXECUTOR_NPROCS is not defined, it will default to the number of created executer threads, so each executer thread will have an entire core. [3]

The third hyperparameter is called TD_EXECUTERS_PER_DEVICE, which will specify how many executer threads should be used for each GPU [25]. With additional executor threads, the computation and data transfer can be overlapped to speed up the overall runtime. As described in 3.1.1, page-locked memory is required for GPU-driven offloading, which is utilized by multiple executer threads for asynchronous data transfer. If this hyperparameter is set to a value greater than one, then 2 threads per GPU need to be placed with TD_MANAGEMENT. If this parameter is not specified, it defaults to one.

The last hyperparameter is the load balancing algorithm. The two possible load balancing algorithms were described in 3.2.2 and in 3.2.2. The default algorithm is the fine-grained algorithm, which is executed permanently. The coarse-grained algorithm needs to be called in the source code, which will make the program run the coarse-grained algorithm for the next five iterations and then use the fine-grained algorithm as it usually does. The fine-grained algorithm can also be deactivated entirely by setting the environment variable SKIP_FINE_GRAINED_SCHEDULING to a non-null value. [3]

Although it is technically not a hyperparameter, pinned memory will be considered as a hyperparameter since a programmer can switch easily from using unpinned memory to pinned memory. To achieve that, one has to use calls like omp_malloc() instead of malloc() and will get a pinned memory address.

Static load is a workload that is computed on the node on which it was created, so it is not migratable. A static load can be part of the use case, but it can also be used to tune the performance. Therefore, static loads are also considered as a hyperparameter for these test cases.

3.3. Existing Benchmarks

In the work of Schmitz et al. [25], four benchmarks and a case study were conducted. This is so far the only evaluation of the tool targetDART. This section will briefly explain how these benchmarks were conducted to provide important results that are known so far.

The benchmarks of Schmitz et al. [25] were conducted on eight GPU nodes of

Claix23 [5] with a custom LLVM build version 20.1.5 as compiler and MPICH version 4.2.2, CUDA runtime version 12.6, and a CUDA driver in version 565.57.01. Reference programs used a custom-built LLVM of the same version, but targetDART was deactivated as a plugin. As a benchmark, they used dense matrix multiplication, whereby the dimensions of the matrices were always 8000×1000 and 1000×1000. They generated 100 tasks on each node, with one task being a single matrix multiplication and all tasks were assigned to the same targetDART device. For each benchmark, multiple targetDART devices were used to compare them.

3.3.1. Weak Scaling

The first benchmark is a weak scaling test, where the number of GPUs increases, while the workload per GPU remains constant. For each GPU, 25 tasks were dispatched, which equals 100 tasks per Claix23 GPU node. The experiment was conducted two times, with the targetDART device `TARGETDART_ANY` and with the device `TARGETDART_GPU`.

The results show that the scheduler and receiver thread induce an overhead of about 1.5%. Scaling the number of GPUs induces more overhead of about 2%. This scaling behaviour is very good and was likely achieved because of the initial load balancing. Every node started with the same number of tasks to process, so the scheduler thread probably did not need to migrate tasks to other nodes very often. The load balancing algorithm is also implemented in a non-blocking way, which means that the execution is not paused for scheduling. Therefore, the scheduling across nodes and the task migration likely did not cause the 2% overhead that is induced by scaling the number of GPUs. The results also show that the targetDART device `TARGETDART_ANY` is faster than just using the GPUs.

3.3.2. Overhead Against OpenMP

For this benchmark, the task size was increased. This means that the dimensions of the input matrices were changed. For the size X, the dimension of the first matrix was changed to $8000 \times X$ and the dimensions of the second matrix to $X \times 1000$. The size was increased from 100 to 10000. Multiple targetDART versions and two OpenMP baselines were used for this measurement, in order to measure the overhead. The task distribution was again the same, so the scheduler and receiver thread probably did not have to migrate any tasks.

The results show that the scheduling did not induce much overhead on a perfectly balanced scenario, like in the previous measurement. Compared to the OpenMP baseline, targetDART is about 0.2-0.25 seconds slower for a small task size, which is likely because of the initialization of the CUDA devices. For bigger task sizes, targetDART is about 5% slower. The computation takes the same time for both versions, but the data management of targetDART and the hand-tuned version is different, which is likely the reason for the increasing overhead.

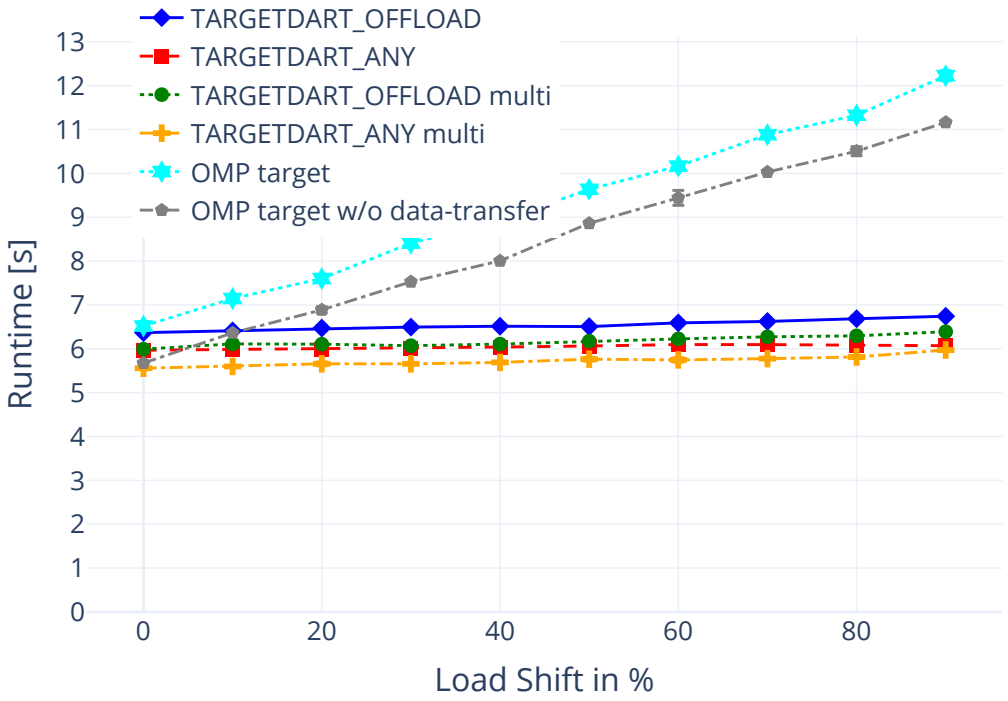


Figure 3.4.: Benchmark results for increasing the initial load shift. The figure was copied from [25].

3.3.3. Application Induced Imbalance

For this benchmark, the number of tasks on each node was different, which creates a load imbalance between the nodes. The test was conducted with 4 nodes and 400 tasks in total, so the overall workload was not changed. The authors introduced the term load shift, which denotes the load imbalance between at least two nodes. For this experiment, a load shift of 10% means that two of the nodes start with 110 tasks and the other two start with 90 tasks. So the load shift was increased and measured with two OpenMP baselines: the basic version and a data optimized version, and with four targetDART versions: `TARGETDART_OFFLOAD`, `TARGETDART_ANY`, `TARGETDART_OFFLOAD_multi` and `TARGETDART_ANY_multi`. The targetDART versions with the `multi` appendix are utilizing multiple executor threads per GPU, which improves the data offloading.

The two OpenMP versions do not utilize load balancing across nodes, which is also clearly visible in the result. With increasing load shift, the runtime of the OpenMP versions increases linearly, while the runtime of the targetDART versions remains almost constant, as Figure 3.4 shows. So the speedup of the two targetDART GPU versions in comparison to the basic OpenMP version is about 1.7. Among the targetDART versions, the `multi` versions always have less runtime than their counterparts.

3.3.4. Hardware Induced Imbalance

For the last benchmark, the authors used eight GPU nodes and for one of them, they lowered the core frequency of all GPUs. The core frequency of the GPU was decreased to simulate power capping, which is utilized in datacenters. The tests were conducted using the same OpenMP and targetDART versions as in the prior benchmark.

The results show a similar behavior to that in the previous benchmark. The runtime of the OpenMP versions increases from about six seconds to almost 30 seconds, which is a slowdown of 4.71× and 4.85×. The targetDART versions have a slowdown ranging from 24% to 35% more runtime. So the overall speedup ranges from 3.46× to 4.1×.

3.3.5. ExaHyPE Case Study

In the work of Schmitz et al. [25], there is also a case study that uses ExaHyPE [23], an engine for solving first-order hyperbolic partial differential equations. ExaHyPE defines the domain in a tree-structured way and refines the tree structure recursively. In order to utilize multiple nodes, it creates subtrees and migrates those subtrees to a different MPI rank, which refines the given subdomain. For GPU offloading, ExaHyPE collects dependencyless tasks and fuses them together into a single task that is offloaded to a GPU. ExaHyPE also has its own load balancer that determines how to split the tree and distribute the resulting subtrees.

For utilizing targetDART, just a few changes were necessary. The authors added a "nowait" to the OpenMP clauses and an "omp taskwait" after the kernel to enable asynchronous execution of the kernel. They also changed the device to which the kernel is offloaded to a targetDART device.

The authors simulate the Tohoku tsunami with ExaHyPE and measure the performance in the average number of cell updates per second, which is depicted in Figure 3.5. They created two scenarios with an imbalanced decomposition of the tree and a well-balanced decomposition of ExaHyPE's load balancer. In contrast to the previous benchmarks, there is just one OpenMP version for the case study. The figure shows that all targetDART versions improved the performance significantly over the OpenMP version for both scenarios. Among the targetDART versions, both versions that only utilize GPUs outperform the other versions, whereby the GPU version that uses multiple executor threads performs slightly better in both scenarios. For the ill-balanced scenario, the GPU versions have a speedup of about 10× compared to the OpenMP baseline. For the well-balanced scenario, the GPU version achieves a speedup of 1.59×. Surprisingly, all targetDART versions perform better at the imbalanced scenario than at the well-balanced scenario. The reason for this might be that the overhead by the load balancing algorithm of ExaHyPE induces more overhead than targetDART.

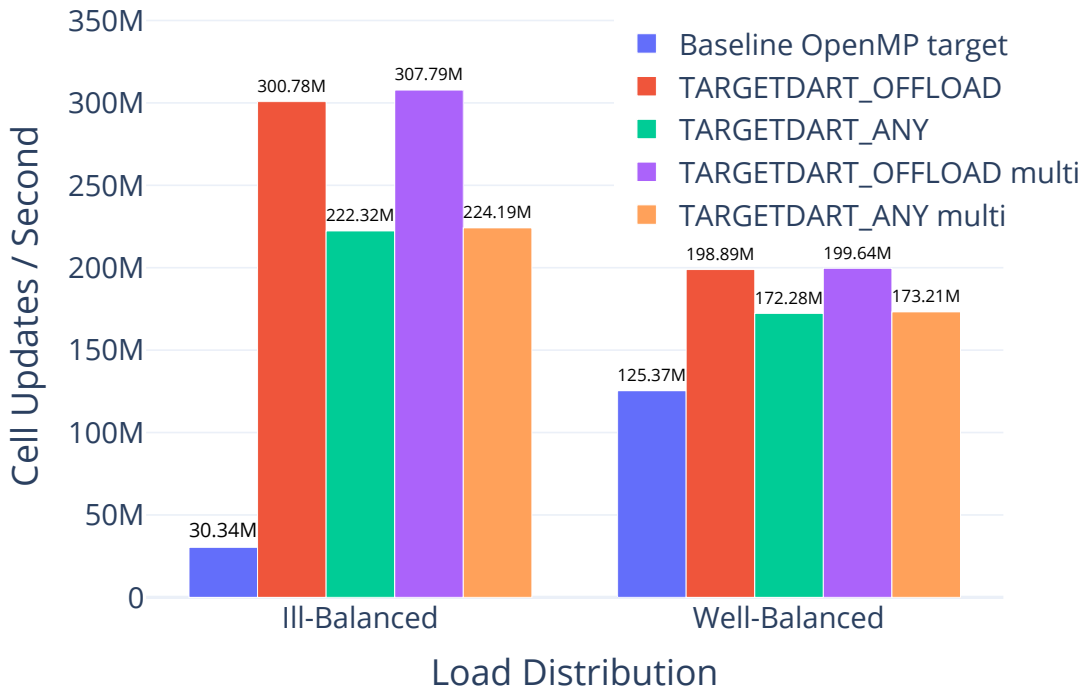


Figure 3.5.: Benchmark results from using targetDART on ExaHyPE. The figure was copied from [25].

4. Benchmarks

The tests of this thesis aim to supplement previous work and testing of targetDART. Furthermore, the results of the tests aim to provide guidance on how to use targetDART. First, the experimental setup is described, including the source code, the hardware, tools like MPI and the environment that was used to set up the benchmarks. Secondly, this chapter will define a "base scenario". This base scenario is a setup of the used parameters, including the hyperparameters. All test cases will be derived from that base scenario by changing a single aspect, like one hyperparameter, to observe the impact of that aspect. The derivation from the base case of each benchmark will be described. This thesis will distinguish between benchmarks that test for which use cases targetDART is suitable and benchmarks that test the impact of hyperparameters. For all benchmarks, the results of the benchmark are presented after the description and are discussed afterward. At the end of the use case benchmarks and at the end of the hyperparameter benchmarks, the implications of the results are concluded. Based on the conclusions, guidelines for the use cases and hyperparameters are proposed.

4.1. Setup

This section briefly describes what hardware and software tools are used to conduct the benchmarks. Furthermore, there are also some settings in the environment that are used to increase the stability of the measurements, which are explained in this section. There is also a short description of the source code that is used to test targetDART. At the end, a base scenario is presented. For each benchmark, the base scenario is used and a single aspect or parameter is changed to see the impact that change has.

Hardware: The benchmarks are conducted on Claix23 nodes with two Intel Xeon 8468 Sapphire Rapids with 48 cores each and 96 cores combined. The nodes are connected via NDR Infiniband in a fat tree topology. Some of the tests are conducted on CPU-only nodes with 256 GB memory and some on GPU nodes with 512 GB memory and four NVIDIA H100 96 GB HBM2e GPUs. [5] On which nodes the benchmarks are conducted depends on the need for GPUs. The figure in Figure A.1 describes the topology of a GPU node. In this figure, the GPUs are described as "cudaX", whereby "X" can be a number between 0 and 3 and the InfiniBand connections as "Net ib0" and "Net ib1". Both are PCIe devices and are mapped

to NUMA nodes, which are also depicted, as well as the CPU cores that belong to those NUMA nodes.

Software/Tools: The OS on the nodes is Rocky Linux 9.6, which utilizes Slurm as a workload manager. For targetDART, a custom LLVM[1] is used together with MPICH version 4.2.2, the CUDA runtime version 12.6 and CUDA drivers version 570.172.08. For the CPU reference, a custom build of the clang compiler version 17.0.6, for compatibility reasons, and MPICH version 4.2.2 are used. For the GPU reference, a custom build of the clang compiler version 20.1.5, OpenMPI version 5.0.3, CUDA runtime version 12.8.0 and CUDA drivers version 570.172.08 are used.

Environment: For the environment, the tests are executed on nodes that are connected by a single switch to guarantee minimal latency and increase the stability of performance measurements. For using Slurm jobs, one can specify the number of GPUs for the entire job or per node. The second option is always used, but if this parameter is lower than four (a GPU node only has four GPUs), the GPUs are chosen randomly. To prevent this, the benchmarks use all GPUs and select the GPUs needed with the `CUDA_VISIBLE_DEVICE` environment variable. For benchmarks that only required one GPU per node, the GPU 1 (on NUMA node 2, as in Figure A.1 depicted) is used.

Source Code: The benchmarks use a program that generates tasks, which consist of a single dense matrix multiplication. After all matrix multiplication kernels are executed, the program checks if all results are correct. The program is compiled with the optimization flag `-O3`. The references that do not utilize targetDART and are used as comparison consist of similar code that just uses unpinned memory and changes the OpenMP pragmas. The references are also compiled with the optimization flag `-O3` too. The code of the targetDART version is in Listing A.1, the CPU reference in Listing A.2 and the GPU reference in Listing A.3.

4.1.1. The Base Scenario

The base scenario uses 2 nodes with 100 tasks for each of them. In case the benchmark needs to be tested with load imbalance, the first node gets 150 tasks and the second one 50 tasks. The size of the matrices is fixed to 8000×1000 and 1000×1000. Pinned memory is used for the input and the output matrices, because only then hyperparameters like `TD_EXECUTERS_PER_DEVICE` have an effect. One executor thread per device is used and only one GPU is utilized. Furthermore, the runtime threads are placed like this `TD_MANAGEMENT=24,25,26,27`, so all threads are on NUMA node 2, which also has a GPU and an infiniband port. All generated tasks are assigned to one targetDART device, but the base scenario does not define which, because

[1] LLVM 20 at commit <eff79605c69566ba96c74cd8d25716e0488c2c3b>

Figure 4.1.: Results of the targetDART GPU and ANY versions, as well as the reference results for increasing the load shift. Lower is better.

most of the test cases will test multiple targetDART devices. The environment variable `OMP_NUM_THREADS` is set to 96. For each configuration that is measured for a benchmark, 27-30 measurements are taken. Due to a bug in LLVM, about 1 of 10 measurements is aborted and not remeasured.

4.2. Use case tests

This section introduces the benchmarks for testing the use cases of targetDART. For these benchmarks, various factors that depend on the tasks and the available resources are tested, such as different task sizes or the number of nodes. First, four benchmarks that test the impact of load imbalances in different situations are presented. The following two benchmarks focus on the impact of the task size. The last two benchmarks test the performance for delayed task generation and for different numbers of tasks.

4.2.1. Scaling the Load Imbalance

For measuring the load imbalance, the term load shift as defined in 3.3.3 by Schmitz et al. [25] is used. For both nodes, the load shift increases from 1% in 1% steps. For the targetDART devices `TARGETDART_OFFLOAD`, which is often referred to as the GPU version, and `TARGETDART_ANY`, the maximum load shift is 24% and for the `TARGETDART_CPU` the maximum load shift that is measured is 70%. The CPU device is measured with higher load imbalances because it has more overhead and needs higher load imbalances in order to compete with the CPU reference. For the GPU and CPU versions, there are handwritten references that do not apply dynamic load balancing. These references are tested with the same load shifts as the other version, in order to compare the performance of targetDART with them and evaluate its performance. Every step of the load shifts of all versions (targetDART versions and the references) is measured 9-10 times[2]. This experiment resembles the benchmark described in 3.3.3, but also tests the CPU device of targetDART and compares it to a reference.

Results GPU: The results of the GPU versions and the ANY version of the benchmark are plotted in Figure 4.1. The GPU and ANY versions of targetDART perform similarly, whereby the ANY version slightly outperforms the GPU version. Both of them are faster than the handwritten reference for all load shifts. The runtime of all versions increases with higher load shift. Compared to a 1% load shift, both targetDART versions, GPU and ANY, get a slowdown of about 11%. The reference version gets a slowdown of 21%. For a 1% load shift, the speedup of the GPU version is 1.15× and for the ANY version 1.18×. The speedup for 24% load shift of the GPU version is 1.26× and the ANY version has a speedup of 1.29×.

Results CPU: The results of the CPU versions are plotted in Figure 4.1. For low imbalance, the targetDART version needs significantly more time than the reference at 1% load shift. The runtime of the reference increases with higher load imbalances, until it has about the same runtime as the targetDART version at a load shift of 58%. For load imbalances above 58%, the targetDART CPU version does not outperform the reference, but has about the same performance. The slowdown of the reference at 24% load shift is 22%, so about the same slowdown as the GPU reference for that load shift. For a 70% load shift, the slowdown reaches 64%. The slowdown of the targetDART CPU version for 70% load shift is 6%.

Discussion: For small load imbalances, the reference is supposed to be faster than targetDART, because starting the runtime threads and orchestrating the task execution should provide some overhead, which the references do not have. But the GPU and ANY version of targetDART always outperform the reference. The reason for this is that targetDART barely induces overhead and uses pinned memory

[2] in about 1 of 10 cases, a measurement is aborted because of a bug in LLVM

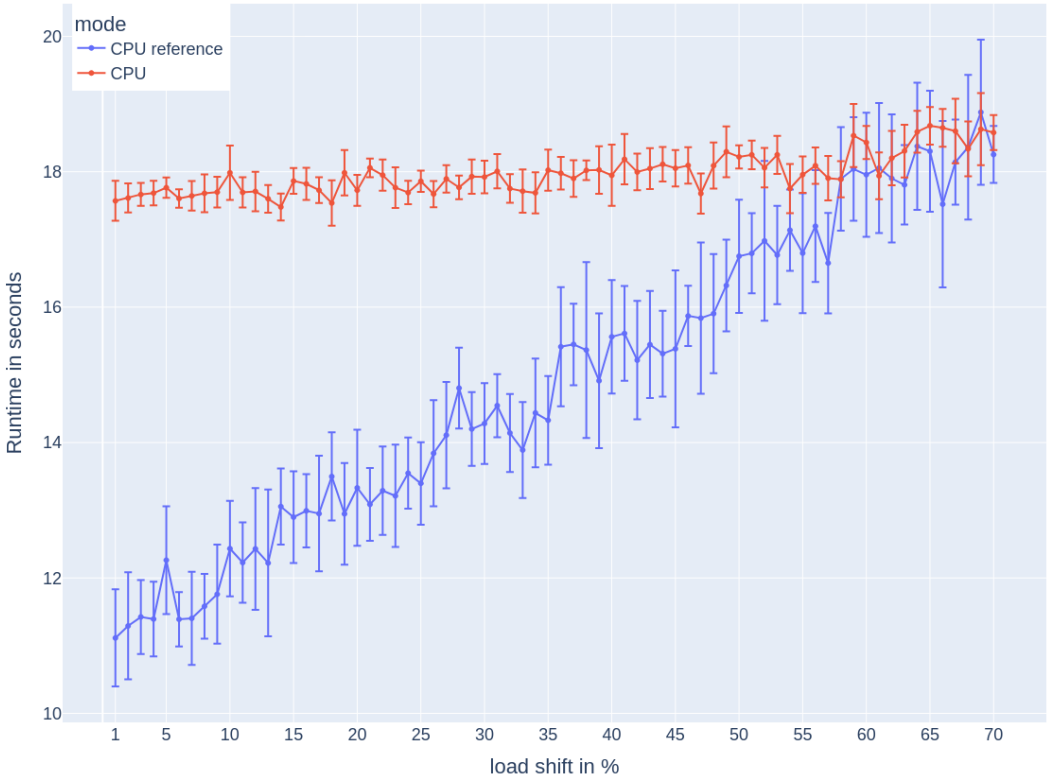


Figure 4.2.: Results of the targetDART CPU version, as well as the reference results for increasing the load shift. Lower is better.

in comparison to the reference. The consequences of this are that the memory of the reference tasks is copied into page-locked memory regions, to offload it to the GPU, as described in 3.1.1, which leads to more runtime overhead. The CPU version of targetDART needs significantly more time than the reference solution. The reason for this is a high overhead, which is induced by the host plugin. In Section 4.2.2, a benchmark compares the host plugin, which targetDART utilizes for kernel execution on the CPU, with targetDART and the CPU reference.

The increasing runtime of all versions results from the increased load shift. For the reference versions, the imbalance causes one node to need more time than the other node, which slows down the entire execution. The targetDART versions have a lower slowdown than the references because they apply dynamic load balancing. The dynamic load balancing also induces overhead for each task that is sent between nodes, which causes the slowdown of the targetDART versions. The scheduler and receiver threads are non-blocking, which means that the execution of tasks is not interrupted for a task migration. So the overhead that is induced by the task migration is small, compared to the overall runtime of the CPU version, which leads to a small relative slowdown. The GPU and ANY versions of targetDART are

much faster, so the overhead is bigger compared to their runtime, which explains the higher relative slowdown. Another factor that increases the slowdown of the GPU and ANY versions is that the tasks are also offloaded to the GPUs.

4.2.2. Overhead of the Host Plugin

In the previous benchmark, the overhead of the targetDART CPU version was, compared to the reference, very high. Since targetDART utilizes the host plugin to execute the kernel on the CPU, this benchmark compares the runtime of the host plugin with targetDART and the CPU reference. To test the performance of the host plugin, a custom LLVM is built, just like for targetDART, but the host plugin is the only plugin that is used. The host plugin LLVM build was used to compile the same code as for the targetDART versions A.1. The performance of all versions is tested with 0% and 50% load shift as it is defined for the base scenario with imbalance.

Results: A graph of the results is in Figure A.2. The host plugin version is for 0% load shift about 14% slower than the targetDART version and for 50% load shift, it is 48% slower than the targetDART version. The CPU reference outperforms the targetDART version just like in the previous benchmark.

Discussion: The host plugin version has a higher runtime than the reference, which indicates a significant overhead for the execution. Even the targetDART version outperforms the host plugin version. Since targetDART utilizes the host plugin, the overhead of the targetDART CPU version is caused by the host plugin. The reason why the targetDART version outperforms the host plugin is that it manages the invocation of the code more efficiently than the host plugin.

4.2.3. Random Load Imbalances

The first benchmark tested the impact of different load shifts for a fixed interval of load shifts. This benchmark tests random load shifts across four nodes, which allows higher load shifts and different scenarios to happen. The random load shift is implemented with the Mersenne Twister of the Python random library [6] as a pseudo-random number generator that outputs the load shift of each node. The average workload remains at 100 tasks per node, so in total, 400 tasks across the four nodes. The Python code for realizing the load balance is in Listing A.4. In the code, the maximum output for the Mersenne Twister is the number of remaining tasks, except for the last node, which gets all remaining tasks. For the first node, the remaining tasks are 400, as this is the total number of tasks. As later benchmarks discovered, a CPU node does not have enough memory to store more than about 250 tasks in its memory, so whenever the Mersenne Twister gives more than 250 tasks, it sets the value to 250 tasks. This also means that the probability of drawing 250 tasks

Test case No.	Task distribution	median runtime	standard deviation
0	100 100 100 100	2.34	0.17
1	53 250 74 23	4.05	0.23
2	250 142 0 8	13.86	0.42
3	95 165 51 89	2.81	0.22
4	94 31 108 167	2.86	0.27
5	249 112 31 8	4.01	0.2
6	250 123 13 14	4.04	0.28
7	129 40 150 81	2.76	0.12
8	90 250 7 53	3.97	0.22
9	185 80 18 117	3.15	0.25

Table 4.1.: A table of the results of the ANY version for the random load imbalance benchmark. Lower is better.

for the first node is higher than for any other number of tasks. Nine configurations of random load shifts with seeds from 1 to 9 were generated and tested with the CPU and ANY devices. For comparison, a reference with an even task distribution is also tested.

Results GPU: The results are listed in 4.1 for the ANY devices and in Table A.1 for the CPU devices. The test cases with better performance always have no node that has more than 185 tasks. To this test cases belong the test cases 0,3,4,7,9 for the ANY device, which all have a runtime of 2.3 to 3.1 seconds. The test cases that perform less well all have one node with at least 249 tasks with a runtime between 3.9 and 13.8 seconds. One outlier is the test case 2 with 13.8 seconds. The test case with the second-highest runtime is test case 1, with about 4 seconds, so the outlier has a runtime that is 3 times higher than the second-highest runtime.

Results CPU: For the CPU device, the test cases 0,3,4,7,9 have runtimes between 11,2 and 23.9 seconds and the other test cases have runtimes ranging from 28.2 to 35.7 seconds. For the CPU version, test cases 4 and 7 manage to outperform the balanced test case.

Discussion: Considering the mechanism of the fine-grained scheduling, the test cases with 250 tasks on a single node are a worst case for the scheduling algorithm. This worst case was discovered in benchmark 4.2.4. If a single node has 250 tasks, it would need to migrate tasks to at least two more nodes in order to load balance the tasks. The remaining nodes balance the workload among themselves and finish before the node with 250 tasks does. This node will then send tasks to all other nodes and will be the only node that sends tasks to different nodes. The fine-grained scheduling algorithm works better if the workload is distributed more equally, so

multiple nodes will send tasks at the same time. So if a major part of the workload is concentrated on one node, it is a worst-case scenario for the scheduling algorithm.

The outlier for the ANY device takes a lot more time than the other test cases, which does not result from a worst-case scenario for the scheduling algorithm. In the benchmark 4.2.8, any node that gets 0 tasks results in some kind of bug, where the runtime increases significantly if the CPU device is used. Since the ANY device also utilizes the CPU device, this bug also occurs for the ANY device. For the CPU device, this bug does not seem to increase the runtime as much as for the ANY device. For the CPU device, the same test case took about 2 seconds longer than the second-worst runtime.

The CPU version with test cases 4 and 7 outperforms the test case with even task distribution. This behaviour still needs further research in order to determine the cause of it, but the only factor that changed was the task distribution, which was likely the cause of the decreased runtime. Since it outperforms the test case with the optimal task distribution, it might cause some kind of optimization if the task distributions of those test cases are chosen.

4.2.4. Load Imbalances with multiple Nodes

The results of the previous benchmark show that targetDART is also able to handle four nodes, but with significant overhead in case one node has about 250 tasks. This benchmark tests if targetDART can handle an arbitrary number of nodes. It uses two to eight nodes and tests the runtime with a balanced task distribution and with an imbalance. In order to regularly increase the load imbalance with the number of nodes, the imbalance is created by assigning 100 tasks to the first node and adding 20 tasks to each of the other nodes that are used. So the formula for the number of tasks for the first node for testing with load imbalance is $80 + N \cdot 20$, whereby N is the number of nodes used.

Results GPU: The results of the ANY version are illustrated in Figure A.3 and the results of the CPU version in Figure A.4. For the results of the ANY version, all tests with an even distribution have a similar runtime. The medians range from 2.3 to 2.38 seconds. Tests with load imbalance have slightly higher runtimes for the test case with two to four nodes. The median of the runtime of the test case with four nodes, which also has the highest imbalance, reaches a runtime of about 2.7 seconds. The test cases with more than 4 nodes and an even higher imbalance have faster-increasing runtimes. The test case with five nodes has a median of 3.18 seconds, which is about 0.5 seconds more than for the test case with 4 nodes. For the remaining nodes, they increase their runtime with each additional node by a time between 0.21 and 0.32 seconds. For the tests with two to four nodes, the runtime increased by 0.03 to 0.09 seconds.

Results CPU: The runtimes of the CPU test cases are all higher than those of the GPU versions. For an even distribution, all test cases have about the same runtime, although the variance of all measurements is higher than for the GPU test cases. For the measurements with imbalance, the median of all tests is between 18.1 and 19.7 seconds. There is no increasing runtime over all test cases with increasing imbalance, as for the GPU test cases.

Discussion: The test cases with imbalance resemble the worst-case scenario that was presented in the previous test case. The median of the runtime increases faster when one node has 180 tasks or more. This provides more insight into how high the load imbalance needs to be in order to behave like the worst case. The load imbalance of the test case with five nodes indicates that the worst-case behaviour appears if one node has more than twice as much workload as all other nodes have.

In contrast to the GPU test cases, the CPU test cases do not show that worst-case behaviour. The CPU versions have a higher runtime than the GPU versions. The additional time of the worst case is induced by the scheduler, which is non-blocking. All CPU test cases have additional runtime that each task needs, which covers the additional time that is caused by the worst case.

Since the test cases with lower imbalance with two to 4 nodes just slightly increased the runtime in comparison to the even distribution, targetDART performs well for that amount of nodes as long as the worst case is not caused. For predicting the scaling behavior for more than four nodes, more research is required.

4.2.5. Scaling the Task size

A different factor that can influence the scaling behavior is the size of the tasks. This benchmark tests the influence the size of a task can have on the runtime. The size of a task in general, is defined by the processing time for a single task. For this benchmark, the task size is defined by the size of the matrices that are multiplied. For the first matrix, the number of columns and for the second matrix, the number of rows is increased from 100 to 2000 in 100 steps. A formula that illustrates the matrices that are multiplied for a task size X, looks like this: $A^{8000\times X} \cdot B^{X\times 1000}$. The measurements are conducted with the GPU and ANY devices for targetDART and with a GPU reference. Each of the devices was tested with a balanced task distribution and with a 50% load shift. A benchmark with a similar setup, but without a load shift, was tested in the work of Schmitz et al. [25].

Results: The results are plotted in Figure A.5. The balanced GPU and ANY version have the best performance and have a lower runtime for all task sizes than almost any other version. The balanced GPU reference has the second slowest version for low task sizes, but scales better than the unbalanced targetDART versions and reaches a similar runtime as the balanced targetDART versions at a task size of 1600. For the task size of 100, the balanced targetDART versions are about 1 second

faster than the balanced GPU version. For the task size of 2000, the GPU reference is about as fast as the balanced GPU version of targetDART and 0.4 seconds faster than the ANY version. The unbalanced GPU reference has the slowest runtime for all task sizes, but it scales similarly to the unbalanced targetDART GPU version. For a task size of 100, the unbalanced GPU reference is 1.1 seconds slower and for a task size of 2000, it is 1.2 seconds slower. For the unbalanced targetDART ANY version, the unbalanced reference has a better scaling behavior, with a decrease in the runtime gap from 1.6 seconds to 0.7 seconds.

The unbalanced targetDART versions scale differently. The unbalanced ANY version performs better than the unbalanced GPU version for small task sizes, but with increasing task size, the gap between the two versions gets smaller until they have about the same runtime for a task size of 1100. For the higher task sizes, the GPU version has lower runtimes than the ANY version. The gap between those versions is 0.4 seconds for a task size of 100, with the ANY version being faster and 0.47 seconds for a task size of 2000, with the GPU version being faster.

Discussion: The balanced targetDART versions have the best performance. Despite having a high overhead for small tasks, the GPU reference has better scaling behaviour with less runtime increment per task size. For the unbalanced versions, the reference has the worst runtime but manages to scale better than the ANY version, similar to the targetDART GPU version. A possible root cause for the targetDART ANY version is that the CPU takes longer to process a task than a GPU. The task stealing implementation of targetDART does not consider that an accelerator could finish a certain number of tasks before a CPU finishes a single one. If, at the end of an execution, just a few tasks remain, this allows the CPU to execute a task, even if an accelerator could finish all remaining tasks faster, if it finishes its task earlier than the accelerator. This would increase the runtime and leave the accelerators idle at the end of the execution. This can happen for arbitrary task sizes, but with increasing task size, the execution becomes more compute-bound, which gives GPUs an advantage for this setup. So the time for which the accelerators remain idle would increase with the task size. This effect increases with the overhead that the host plugin induces.

4.2.6. Heterogeneous Task sizes

Heterogeneous tasks are often part of scientific computations. Different types of tasks can also have different processing times, which the scheduler needs to handle. This benchmark tries to simulate different tasks by giving them different sizes and testing them with a balanced and an unbalanced scenario. For the balanced version, all tasks of the first node get a task size of 1500, whereby the task size is defined as in the previous benchmark 4.2.5. To keep the original workload, all tasks of the second node get a task size of 500, so the overall workload does not change. For the unbalanced scenario, the first node gets 150 tasks, but only the first 100 of them have a task size of 1500; the remaining tasks have a task size of 500. The second

node has 50 tasks with a task size of 500, so the overall workload does not change here either. The benchmark is conducted with the targetDART CPU, GPU and ANY versions, as well as a CPU reference and a GPU reference. The code for all versions was modified to implement the heterogeneous task sizes. The code for the targetDART versions is in Listing A.5, the code for the CPU reference in Listing A.6 and the code for the GPU reference in Listing A.7.

Results GPU: Both targetDART versions perform similarly, as shown in Figure A.6. The median of the ANY version outperforms the median of the GPU version in the balance scenario by less than 1%, and for the unbalanced version, the GPU version performs about 1% better. The ANY version has higher variance in its results than the GPU version. Compared to the previous benchmark, the balanced targetDART versions with a task size of 1000 are 14-16% faster than in this benchmark. But both targetDART versions are also 22-28% faster than their counterparts of the previous benchmark with a task size of 1500. For the balanced scenario, both targetDART versions are about 31,9% faster than the GPU reference. For the unbalanced scenario, the GPU and ANY versions are 45-46% faster than the reference. In the previous benchmark, the unbalanced targetDART version is about 4% faster for a task size of 1000. In comparison to the balanced version, the targetDART versions are about 18% slower. This gap is smaller than in the previous benchmark, where the balanced versions are 27% faster than their unbalanced counterparts.

Results CPU: The results of the CPU versions are plotted in Figure A.7. The balanced targetDART CPU version has a higher runtime than the reference, just like for the other benchmarks. Compared to the targetDART CPU version in 4.2.2, it is 15.3% slower in this benchmark. In contrast to all previous benchmarks, the CPU version gets faster by 4% with a load shift. With that, it is just 2% slower than the reference and 8% slower than the targetDART CPU version of the benchmark that measured the overhead of the host plugin.

Discussion: The scheduler of targetDART does not consider the task size when deciding how many tasks are supposed to be migrated. So with a balanced task distribution, the scheduler will decide that no tasks need to be migrated, despite the first node having the majority of the workload. But the scheduler will decide to migrate tasks as soon as the difference in task numbers between the nodes is higher than a predefined threshold. This will inevitably happen because the second node finished its tasks faster. It still causes a delay in the load balancing. This induces overhead and slows down the execution by 14-16%. With that overhead, it still performs better than the balanced versions of the previous benchmark, which have a task size of 1500.

For the CPU version, adding load shift does not increase the runtime, nor does it stay the same; it decreases. The same effect as in benchmark 4.2.4 could prevent an increased runtime for adding load shift, but it does not explain why the runtime

decreases. The reason for this is that after targetDART is started and dispatches the execution of the first tasks, it migrates the first tasks in the queue. On the first node, the tasks with a size of 1500 are generated first and will therefore be migrated to the second node. All tasks with a size of 500 that are generated on the first node will stay there. The task migration occurs immediately since the threshold for the task difference is exceeded. Since the bigger tasks are migrated to the second node, which only has small tasks, it balances the workload better than in the balanced scenario right at the beginning of the execution. The GPU and ANY version might benefit from this, since the gap between the balanced and unbalanced versions is smaller in this benchmark than in the previous one. After some task migrations, the unbalanced scenario will have a better load balance than the balanced scenario. The overhead for the unbalanced version is still bigger, but the balanced version has an overproportional disadvantage through that.

4.2.7. Delayed Task Generation

For some real-world examples, not all tasks are generated at the beginning of the computations. To test which overhead a scenario like this might add to the runtime of targetDART, a benchmark that will generate all tasks in two batches is tested. The first batch is generated at the beginning and the other one after a certain amount of time, which is specified by the last parameter. Each batch generated half of all tasks that are generated on that node. The source code for this benchmark is in Listing A.8. The benchmark was conducted with the targetDART ANY device with a balanced and an unbalanced task distribution. The delays that are tested range from 0 to 2000 milliseconds in 250ms steps.

Results: The results are depicted in Figure A.8. For all delays, the balanced version outperforms the unbalanced one by 8-29%. The tests with no delay between the two task generations have the lowest runtime. The first step with 250 ms induces the highest overhead with 26% or 0.6 seconds for the balanced scenario and 12% or 0.4 seconds for the unbalanced scenario. For the unbalanced scenario, that delay causes the highest runtime, but for the other delays of up to 1750ms, the median runtime stagnates between 3.2 and 3.5 seconds. With a 2000ms delay, the runtime of the unbalanced version increases by 8.6% to its global maximum.

The runtime of the balanced version stagnates for delays between 250 and 1000 ms and has a local minima for 1250 ms, which is 13.4% faster than with a 1000ms delay. Starting at a 1500ms delay, the runtime of the balanced version increases with higher delays to its global maximum at a 2000ms delay.

Discussion: Generating and queuing half of the tasks induces overhead. The reason why overhead is induced by this is unknown. The overhead for the unbalanced version is lower than for the balanced version when the delay is introduced. The reason for this could be that the overhead, which is induced by load balancing, over-

laps with the overhead that is induced by the delayed task generation. For a delay of 1500ms and for 1750ms, the balanced and unbalanced versions start to get more overhead for higher delays. Since the runtime without any delay is about 2.3 seconds for the balanced and 3 seconds for the unbalanced version, the first half of the tasks finishes its execution in about half of the time. So, targetDART finishes before the second half is generated and the runtime increases for higher delays almost linearly. For smaller delays, the runtime remains for both versions almost constant, which indicates that the overhead is not related to the duration of the delay, as long as the second half is generated before the execution of the first half finishes.

4.2.8. Scaling the Number of Tasks

The final benchmark of testing targetDART for certain use cases is to scale the number of tasks. The number of tasks is scaled from 1 to 30 by adding a single task each step. The maximum was set to 30 tasks because of limited computation resources. The benchmark is tested with the CPU, GPU and ANY versions of targetDART and four references. For those three, all tasks are generated on the first node. The four references consist of two CPU references and two GPU references. For both devices, one reference generates all tasks on the first node, like the targetDART versions. The other reference has a balanced task distribution. Each configuration is measured 9-10 times[3]. Due to a bug in TargetDART that increases the runtime drastically for the CPU and ANY device if a node starts with zero tasks, each node that would start with zero tasks starts instead with one task (all versions). In Figure A.9, the runtimes for the CPU and ANY version are plotted with the same environment as this benchmark, but with the second node starting with 0 tasks, so with the bug. The consequence of this is that the overall workload is increased a little bit. For the references, this barely makes a difference. For the targetDART version, this effectively would shift the graphs on the x-axis, but it will not affect how it scales with an increasing number of tasks.

Results GPU: For the GPU references and the targetDART GPU and ANY version, the results are depicted in 4.3 and for the CPU results, in 4.4. All versions have for one task about the same runtime, except the ANY version, which starts with an overhead of 0.2 seconds compared to the other versions. While the runtime of all GPU versions increases linearly, the runtime of the ANY version remains constant for up to ten tasks, then it starts increasing like the other versions. The balanced GPU reference outperforms every other version, unlike the unbalanced reference, which is outperformed by every other version, after the targetDART ANY version surpasses it at 14 tasks. The targetDART GPU version performs better than the unbalanced reference, but worse than the balanced one. The ANY version surpasses the unbalanced reference at 14 tasks, but never performs better than the targetDART GPU version or the balanced reference. For 30 tasks, the GPU version

[3]in about 1 of 10 cases, a measurement is aborted because of a bug in LLVM

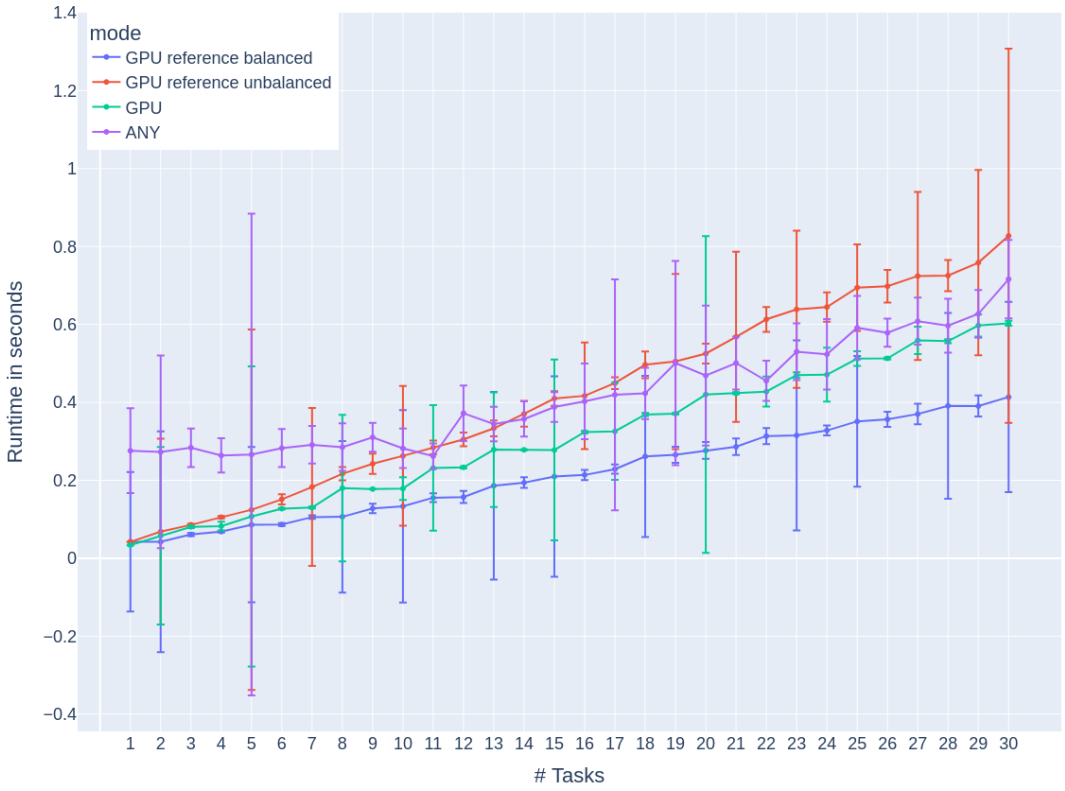


Figure 4.3.: Results of the targetDART GPU versions, as well as the reference results for increasing the number of tasks. Lower is better.

was 45% slower than the balanced reference, but 37% faster than the unbalanced reference. The ANY version is for 30 tasks, 73% slower than the balanced reference and 15% faster than the unbalanced reference.

Results CPU: From the CPU versions, all of them start with about the same runtime for one task. All of them scale linearly, with the balanced reference outperforming the other two. The targetDART CPU version performs similarly to the unbalanced CPU reference for all numbers of tasks. For 30 tasks, the balanced reference is 79% faster than the other reference and the targetDART version. The runtime medians of the unbalanced reference and of the targetDART version have less than 1% difference for 30 tasks.

Discussion: The ANY and the GPU versions of targetDART perform better than the unbalanced reference, so they provide a benefit in the execution. The targetDART GPU version always outperformed the unbalanced reference, so it can be used to gain a performance benefit independently of the number of tasks. The ANY version outperformed the unbalanced reference with more than 14 tasks, so for

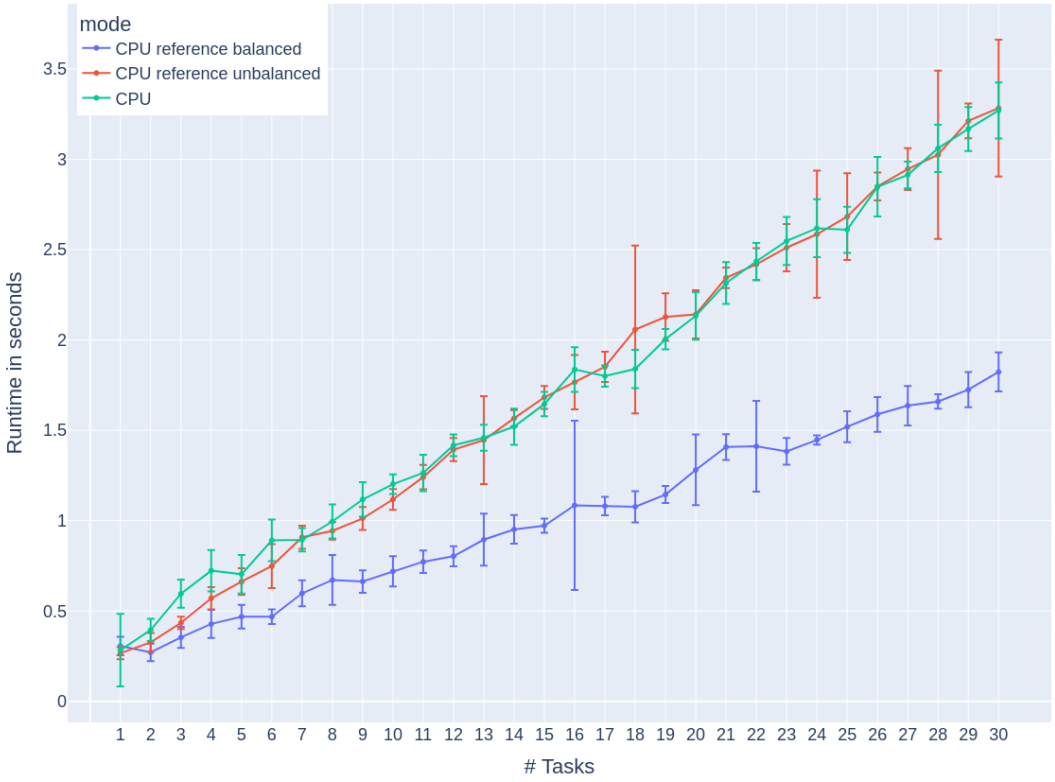


Figure 4.4.: Results of the targetDART CPU version, as well as the reference results for increasing the number of tasks. Lower is better.

more than 14-15 tasks are required for the ANY version to speed up the execution. The ANY version did not outperform the GPU version, which contradicts previous benchmarks. In the first benchmark, for example, the ANY version outperformed the GPU version of targetDART independently of load shift. In this benchmark, the load shift is very high and the GPU version is outperforming the ANY version instead. The results of the first benchmark 4.2.1 indicate that the GPU version does not outperform the ANY version in this benchmark because of the load shift, but because of the number of tasks. Since the load shift is higher in this benchmark, more testing is required to be sure what caused this anomaly.

The CPU version performs similarly to the unbalanced reference. Because of the additional task, which was added as described earlier, the targetDART CPU version might perform a bit better than the unbalanced reference, since the additional task barely influences the reference. But it does not influence the scaling behaviour, which remains the same. So it does not decrease the runtime significantly to use the CPU version for any number of tasks.

4.2.9. Guidelines for Use Cases

In this section, the conclusions on the results of the previously presented benchmarks are listed. Based on those conclusions, guidelines that indicate in which situations the performance can be increased using targetDART are derived. The guidelines will also point out in which situations, which targetDART devices will increase the performance more.

Conclusions on benchmark 4.2.1: The results show that the use of the GPU and ANY versions of targetDART can increase the performance and that it performs better than a reference version with load imbalance. Therefore, these two versions fulfill the purpose of targetDART. For the CPU version of targetDART, the overhead is a lot higher. If computations can just be conducted on nodes with CPUs, targetDART should not be used since it performs worse for load shifts below 58% and has about the same performance as the reference for load shifts above 58%. It is unknown if the targetDART CPU version would outperform the reference for higher load imbalances.

Conclusions on benchmark 4.2.2: The runtime of the version with the host plugin is longer than the runtime of the targetDART version. This shows that the overhead is generated by the host plugin. The code optimizations of the host plugin are less advanced and therefore, the execution consumes more time.

Conclusions on benchmark 4.2.3: targetDART is also able to load balance the load efficiently with four nodes. In case a single node has about 250 tasks of the workload, targetDART's scheduling algorithm will perform significantly worse because putting that much workload on a single node is the worst-case scenario of the algorithm. For other task distributions, targetDART performs similarly to the first benchmark.

Conclusions on benchmark 4.2.4: The worst case is likely to occur if a single node has about twice as many tasks as the other nodes have. As the results of the previous benchmark show, the remaining nodes do not need to have the same workload. For a small number of nodes, targetDART induces little overhead, compared with a perfectly balanced test case. In previous research to targetDART [25], some benchmarks were conducted on eight nodes without the worst case for load balancing. These benchmarks show good scaling behaviour, so targetDART also induces little overhead for eight nodes. These data samples indicate that targetDART is able to scale with an arbitrary number of nodes.

Conclusions on benchmark 4.2.5: The GPU version of targetDART shows resilient scaling behaviour, in contrast to the ANY version. For small task sizes, the ANY is still very fast, but the scaling behaviour indicates that for a certain task

size, the reference could outperform the ANY version. For real-world scenarios, the task sizes can be quite large, making the ANY device less attractive.

Conclusions on benchmark 4.2.6: The GPU and ANY versions of targetDART perform better than their counterparts in the previous benchmark with a task size of 1500. There is still some more overhead of about 14-16% compared to other measurements, but targetDART is able to compensate for the load imbalance that is induced by heterogeneous tasks. The CPU version has similar overhead because of the heterogeneous tasks, but it is still slower than the reference. The targetDART versions are also suited for use cases with heterogeneous tasks, although the CPU version still just provides a benefit in cases of very high load imbalances. All targetDART versions benefit in the unbalanced scenario from the order of the task generation, which causes the CPU version to perform better than in the balanced scenario. To estimate the overhead caused by heterogeneous tasks with a load imbalance, more research is recommended.

Conclusions on benchmark 4.2.7: Since the regular use case of targetDART will be that the workload is unbalanced, the expected overhead for a delayed task generation is not very high, with a constant overhead of about 400ms. A possible cause for the overhead could be the scheduler. Previous benchmarks show that increased task size reduces overhead that is induced by the scheduler, like for the worst case scenario, which had less overhead for the CPU device in 4.2.4. Therefore, a promising benchmark for further research would be to test if the overhead is reduced for increased task sizes.

Conclusions on benchmark 4.2.8: The targetDART OFFLOAD device provides, for any number of tasks, a benefit, while the ANY device does for a number of tasks that is greater than 14, which is not very much. Since the GPU version outperformed the ANY version in this benchmark and the tested numbers of tasks that were tested in this benchmark, the OFFLOAD device is recommended for a small number of tasks. For higher numbers of tasks, other benchmarks like 4.2.1 indicate that the ANY device performs better. More research is required to understand some anomalies of this benchmark. Since the number of tasks depends on the granularity of the tasks, testing the number of tasks and the task size combined would make sense and is recommended for further research.

Guidelines:

- Do not use the CPU device of targetDART. In Benchmark 4.2.1, the load shift for which targetDART is equally fast to a reference was about 58%. It is unknown if the targetDART CPU device performs better for high load shifts above 70% and therefore the CPU device is not recommended for load shifts of that range either. The overhead for the CPU device is caused by the host plugin.

- Avoid putting a lot of tasks on a single node, so that it has about twice as many tasks as all other nodes. The scheduler of targetDART performs better if the workload is distributed more evenly.
- Avoid giving a node 0 tasks, because this will increase the runtime drastically as mentioned in 4.2.8.
- targetDART scales efficiently for different task sizes and number of tasks and therefore supports arbitrary task granularity.
- For small task sizes, the ANY device of targetDART is recommended, since it outperformed the OFFLOAD device in 4.2.5.
- For a small number of tasks, the targetDART OFFLOAD device is recommended, because it performs better than the ANY device in 4.2.8.
- targetDART can operate efficiently on 2-8 nodes, but might also be able to work efficiently with more nodes.
- Using heterogeneous tasks with different task sizes and using delayed task generation induces a small overhead. This overhead should be avoided, but since it is a low overhead of 14-16% for heterogeneous tasks and 400ms for the delayed task generation, targetDART is still suitable for use cases where these conditions are inevitable.

There are still a lot of anomalies that cannot be explained with the current knowledge about the behavior of targetDART. To understand these anomalies, more research is required, as well as testing targetDART for more scenarios. Another reason for conducting more research is that for some benchmarks like 4.2.4 or 4.2.6, the overhead was reduced when using the CPU device. A possible root cause for this could be that the task size is automatically increased for executions on the CPU, because the CPU cannot process the tasks as fast as the GPU can. Therefore, retrying some of the benchmarks with manually increased task size could reduce the overhead in some cases, which would increase the performance for coarse-grained tasks.

4.3. Testing the Hyperparameterspace

This section presents the benchmarks that tested various hyperparameters of targetDART. For each hyperparameter, there is a benchmark, which tests the impact of the hyperparameter. Each section will explain the benchmark, present the results, discuss the results and conclude on the impact of that hyperparameter. All hyperparameters of the Section 3.2.3 are tested, except `TD_EXECUTOR_NPROCS` because the effect of that hyperparameter can also be achieved by setting `TD_MANAGEMENT` accordingly and the `OMP_NUM_THREADS` because it is known that this parameter should

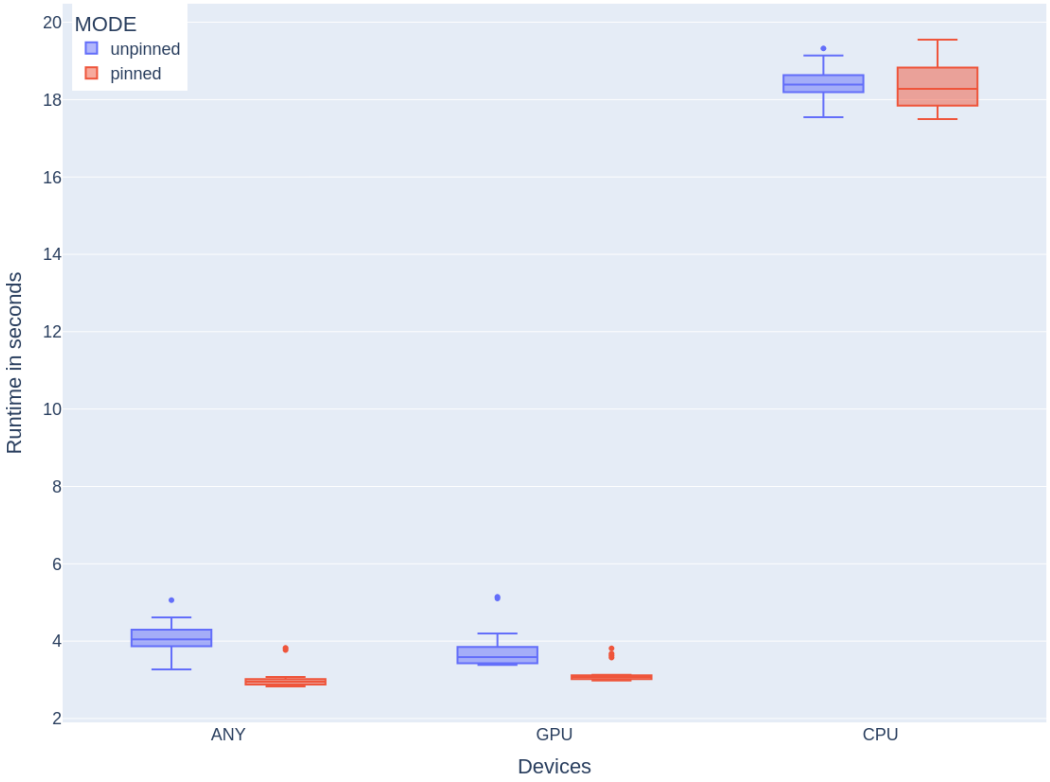


Figure 4.5.: Testing each targetDART version with pinned and with unpinned memory. Lower is better.

be set to the number of cores. At the end, a best-case, a naive-case and a worst-case scenario in terms of hyperparameters are tested and compared, to see what impact the hyperparameters combined have.

4.3.1. Pinned Memory

The first hyperparameter benchmark tests the impact that pinned memory or page-locked memory has. As explained in 3.1.1, if data is offloaded to an accelerator and the accelerator manages the data transfer, then it needs to be stored in pinned memory. If the data is not stored in such memory, it is copied to a pinned memory section, which creates overhead. The benchmark is tested with the ANY, the OFFLOAD and the CPU devices of targetDART.

Results: The results are plotted in Figure 4.5. For the GPU and ANY versions, the medians of the runtimes decrease, as well as the variance of the runtime, which is indicated by the flattened boxplots in the figure. The ANY version with pinned memory has a speedup of 1.369× compared to the unpinned version. For the GPU

version, the speedup is 1.168×.

Results CPU: The performance of the CPU version barely changes. The median of the runtime decreases by less than 1% and 100ms in total. The variance of the runtimes also hardly changes, as shown by the upper and lower fences in the figure.

Discussion: For the CPU version, the pinned memory does not have a significant impact and does not provide any performance benefit. For the GPU and ANY versions, pinned memory reduces the runtime significantly. The reason for this is that in order to copy data to a GPU, the data has to be in a pinned memory region. If it is not stored in pinned memory, it is copied into one before the data is offloaded to the GPU. In our setup, the input data consists of two matrices, which are copied to a GPU every time a task is executed on a GPU. If the matrices are stored in unpinned memory, they are copied for each task execution on a GPU to pinned memory before they are offloaded. Therefore, for all tasks of the GPU version, the input needs to be copied. For the ANY version, the memory does not need to be copied to pinned memory if the task is executed on a CPU. Surprisingly, the speedup for using pinned memory is higher for the ANY version than for the GPU version. The reason for that needs further investigation.

4.3.2. Thread Placement

In this benchmark, different placements for the runtime threads, which are explained in Chapter 3.2.1, are tested. To place the runtime threads on certain cores, the hyperparameter `TD_MANAGEMENT` was used. The tests are only conducted with cores from the first NUMA node. Tests where different NUMA nodes are used are conducted in the following benchmark 4.3.3. In total, this benchmark contains six configurations, which are tested:

1. All runtime threads on one core (`TD_MANAGEMENT=0,0,0,0`).
2. All threads on different cores (`TD_MANAGEMENT=0,1,2,3`).
3. Scheduler and receiver thread on different cores, but the executor thread for the GPU and CPU are placed together (`TD_MANAGEMENT=0,1,2,2`).
4. Scheduler and receiver thread are placed together and each executor thread is placed on its own core (`TD_MANAGEMENT=0,0,1,2`).
5. The scheduler thread and all executor threads are placed on the same core. The receiver thread is placed on a different core (`TD_MANAGEMENT=0,1,0,0`).
6. The receiver thread and all executor threads are pinned to the same core and the scheduler thread has its own core (`TD_MANAGEMENT=0,1,1,1`).

Test case Nr.	median of the runtime	standard deviation
1	3.26	0.51
2	2.93	0.13
3	2.99	0.33
4	2.76	0.05
5	3.17	0.24
6	3.06	0.41

Table 4.2.: The median runtime and the standard deviation of the benchmark 4.3.2. The test case number refers to the enumeration in 4.3.2. Lower is better.

These test cases are not all possible combinations for the placement of the runtime threads. The benchmark was just tested with the targetDART ANY device. Each configuration is measured 9-10 times[4].

Results: The results are presented in Table 4.2 with the indices of the previously presented test cases. The best performance is achieved by test case 4. It does not just have the lowest median runtime but also the lowest standard deviation. The second-best result has test case 2, which places all threads on different cores. It is about 6% slower than test case 4 and also has a very low standard deviation. The third best result has test case 3, which does the same as test case 2, but it places the executor threads on the same core. This test case also has a standard deviation that is nearly three times higher than that of test case 2 and it is about 2% slower than test case 2. The worst-performing test case is test case 1, which puts all threads on a single core. The runtime of that test case is 18% slower than that of the test case with the best performance and has a very high standard deviation.

Discussion: The best performance has the testcase that places the scheduler and receiver thread together. Therefore, this configuration is recommended. This benchmark used the fine-grained scheduling algorithm, as defined by the base scenario. For this algorithm, it is impossible for a node to migrate some of its own tasks to a different node and simultaneously receive tasks from a different node. This means that sending tasks and receiving tasks never happen on one node at the same time. This explains why test case 4 performs well. The receiver thread is also responsible for receiving the results of migrated tasks, which can happen while sending tasks, but it does not seem to induce a lot of overhead. Test case 2 outperforms test case 3, which leads to the conclusion that it does not provide any performance benefit to place executor threads on the same core.

[4]in about 1 of 10 cases, a measurement is aborted because of a bug in LLVM

4.3.3. NUMA nodes

After testing, which placement of threads on a single NUMA node achieves the best performance, this benchmark tests which NUMA node achieves the best performance. Since PCI devices are assigned to different NUMA nodes, as explained in Chapter 3.1.2, and the NUMA nodes span across two CPU sockets, the NUMA node that is used could potentially influence the performance. A figure of a Claix23 GPU node is in Appendix A.1. The PCI devices that are suspected to influence the performance are the GPUs and the InfiniBand connections, which are used for the internode communication. The GPU that was used is GPU 2, according to the base scenario.

The configuration that had the best performance in the previous benchmark is used and tested on all eight NUMA nodes. The cores to which the runtime threads are pinned will change for each NUMA node that is tested, but the scheduler and receiver threads are placed together and all other threads are placed on their own core. The benchmark was conducted with just the ANY device of targetDART.

Results: The results of the benchmark are plotted in Figure A.10. The NUMA node that achieves the highest performance gain is NUMA node 3, which has no PCI devices assigned to it. It performs 1-3% better than all other NUMA nodes of the first socket and 3-5% better than all NUMA domains of the other socket. All NUMA nodes of the first socket perform better than the nodes of the second socket. The variance of all NUMA nodes is similar.

Discussion: The reason why NUMA node 3 has the best performance needs further investigation. A possible root cause for the increased performance of all NUMA nodes of the first socket is that the GPU, which is used, is located on the first socket. That would also lead to the conclusion that the NUMA node to which the GPU is connected performs best, but it is outperformed by node 3. This means that there is some anomaly that makes node 2 slower than node 3. To determine the reason for that anomaly, further investigation is required.

4.3.4. Multiple Executor Threads

The hyperparameter `TD_EXECUTORS_PER_DEVICE` is tested in this benchmark. The usage of multiple executor threads enables targetDART to overlap the data offloading and computation for tasks that are offloaded to accelerators. A requirement for this is that the data needs to be stored in pinned memory, which is defined in the base scenario. The benchmark is conducted with the targetDART OFFLOAD and ANY devices. For both devices, the number of executor threads is scaled from one to four.

Results: The results are plotted in Figure 4.6. For both GPU and ANY version, the runtime decreases when using multiple executor threads. Using two executor

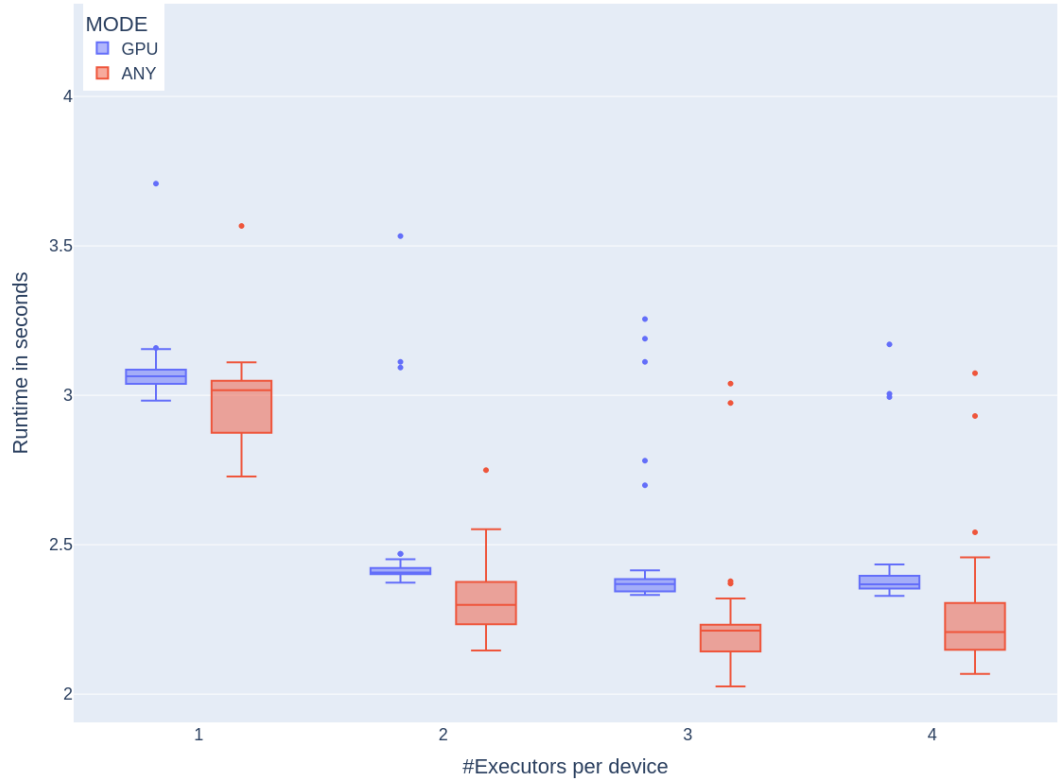


Figure 4.6.: Testing the OFFLOAD and ANY devices of targetDART with multiple executor threads. Lower is better.

threads instead of one decreases the runtime a lot, while using three or four executor threads just slightly improves the performance compared to two executor threads. Compared to one executor thread, the GPU version is executed 27.2% faster with two executor threads, 29.3% faster with three executors and 29.4% with four. For the ANY version, using two executor threads instead of one decreases the runtime by 31.2%. For three executors, the runtime decreases by 36.3%, compared with one thread, and for four executors, the runtime decreases by 36.6%.

Discussion: Using two executor threads instead of one creates a big performance benefit. Using three executor threads instead of two creates a much smaller benefit of about 2.1% for the GPU version and 5.1.% for the ANY version. For the use of four executors, the performance benefit is almost negligible, with below 1% for both versions.

The impact for the ANY version is higher, although it should also decrease the performance of the CPU because fewer cores can be used for computation. But instead, the performance gain is higher for the ANY version. The reason for this is unknown.

4.3.5. Setting OMP_NUM_TEAMS

This benchmark tests what impact the hyperparameter `OMP_NUM_TEAMS` has. This parameter sets the number of teams that are spawned for code executions on the GPU. It is tested with not being defined, like in the base scenario, and with 512, 1024, 2048 and 4096. The benchmark is conducted with the ANY device.

Results: The results of the benchmark are represented in Figure A.11. If the environment variable is undefined, targetDART performs better than if the variable is set to any of the tested values. Setting the hyperparameter to 512 caused the worst performance, which has a 15.2% slower median than the version where it is undefined. For the values 1024,2048 and 4096, the performance was similar, with the medians being between 3.07 and 3.1 seconds, which is 5.2-6.2% slower than the version, where it is undefined. The variance for all configurations is very similar.

Discussion: Leaving the parameter undefined accomplishes the best results. The reason for this needs further research. Setting the parameter to 512, results in the worst performance. A reason for this could be that this value is too small to fully utilize the GPUs For the values 1024, 2048 and 4096, the runtimes stagnate on values that are 5-6% slower than the values for leaving the parameter undefined. Possible causes for that could be that the CUDA plugin scales the value for `OMP_NUM_TEAMS` down if it is too large.

4.3.6. Static Loads

This benchmark tests the impact that statically assigned tasks have on the runtime. With the `TARGETDART_X_LOCAL` device, as described in 3.2, the user can assign tasks, which will not be migrated, to a node. This benchmark used two binaries, which generate the local and non-local tasks in a different order. Both generate a number of tasks with the `TARGETDART_X_LOCAL` and generate more tasks with a non-local device. The binaries are used for a balanced and an unbalanced scenario, where on both nodes the total number of generated tasks is the same as defined in 4.1.1. If, for example, 5 tasks are statically assigned to each node with the `TARGETDART_ANY_LOCAL` device, for the balanced scenario, 95 more tasks are generated with the `TARGETDART_ANY` device, making a total of 100 tasks on each node. The source code of the first one is in Listing A.9. It generates the static tasks first and generates all other tasks secondly, so this version will be called the static-first version. The source code of the other version is in Listing A.10 and it generates the static tasks after the other tasks. Since the task generation order is reversed, this version will be referred to as the reversed version. The number of static tasks for this measurement will be between 5 and 50 and will be incremented by 5 tasks. The benchmark is tested with the ANY and the GPU device and with a balanced scenario and an unbalanced one. For comparison, the balanced and unbalanced versions are also executed with no statically assigned tasks.

Results: The results of the benchmark are plotted in A.12. The unbalanced versions both have higher runtimes than the balanced versions. For the balanced versions, the static-first and the reversed versions both perform similarly. The performance of both versions stagnates with increasing static workload. The two versions often perform better than the baseline for the balanced scenario, but the best result, which is achieved with five static tasks, is just 3.5% faster than the baseline, so both versions are very close to the baseline.

For the unbalanced versions, both runtimes are getting higher with an increasing number of static tasks. In the majority of the measurements, the static-first version slightly outperforms the reversed version. For the static-first and the reversed version, both outperform the unbalanced baseline with just five static tasks and with no other number of static tasks. The static first version manages to decrease the runtime by 2.4% in comparison.

Discussion: Reversing the order of task generation does not provide any performance benefit, since the reversed version performs similarly to the static-first version or slightly worse. The only number of static tasks that was able to outperform the baseline, in the balanced and unbalanced scenario, was with 5 static tasks. Since the measurements are conducted by increasing the number of static tasks always by 5, it requires more measurements to determine the exact optimum. For the unbalanced scenario, some kind of overhead is caused during the execution, if more than 5 static tasks are assigned to each node, because the runtime increases for those measurements. A possible root cause for that overhead could be that the overhead is caused by the scheduler. The perfect load balance could be achieved for all configurations because there were never more than 50 static tasks, so half of the tasks for a perfect distribution.

4.3.7. Different Scheduling Strategies

There are two different scheduling algorithms implemented into targetDART, which can also be combined. This benchmark tests three different configurations for the scheduling algorithm:

1. Default: It will use the fine-grained algorithms, which is the default algorithm.
2. Coarse-grained: It will also test the coarse-grained algorithm, which was explained in Section 3.2.2. The coarse-grained algorithm is called with the function `td_phase_progress()` once, so this algorithm is applied for five iterations.
3. combined: This configuration uses both algorithms combined. So for the first five iterations, the coarse-grained algorithm is used and after that the fine-grained algorithm is applied till the execution finishes.

The different configurations are tested with a balanced and an unbalanced scenario, both of which are tested with the targetDART ANY and CPU device.

Results ANY: The results for the ANY device are plotted in Figure A.13, where "ANY with coarse grained" describes the third configuration, which utilizes both algorithms and "ANY with coarse without fine" describes the configuration that only utilizes the coarse-grained scheduling. For the balanced scenario, the default and the combined configuration perform almost identically with about a 1% difference in the median runtime. The coarse-grained configuration consumes about 2.27× more runtime than the other two configurations. For the unbalanced scenario, the default configuration outperforms the others. The combined configuration takes 31.1% more time than the default configuration and the coarse-grained configuration takes 89.5% more time.

Results CPU: The results for the CPU device are illustrated in A.14. The default configuration outperforms the others for both scenarios. For the balanced scenario, the default configuration is 4.5% faster than the combined configuration and 5.3% faster than the coarse-grained one. For the unbalanced scenario, the default configuration is 3.2% faster than the combined configuration and 6.3% faster than the coarse-grained configuration. The variance of all measurements is higher than for the ANY device. For each configuration, the results have a range of about 2.5 seconds.

Discussion: Using the coarse-grained scheduling algorithm does not provide any performance benefit, nor does the combination of the two algorithms. The coarse-grained scheduling algorithm has a lower complexity than the fine-grained one and should be able to balance the workload for the unbalanced scenario within one iteration. Furthermore, the coarse-grained algorithm also induces overhead for the balanced scenario, where the algorithm should not migrate any tasks. This indicates that the coarse-grained scheduling algorithm does not work as intended.

4.3.8. Best Case vs. Naive Case vs. Worst Case

This benchmark does not intend to test which values for a certain hyperparameter achieve the best performance, but to give a comparison of how a tuned targetDART version performs compared to a non-tuned or badly-tuned version. The benchmark will compare a best case with the best performing hyperparameters, a naive case which barely uses any hyperparameters and a worst case, which uses the worst performing hyperparameters. A table of the used hyperparameters for the best, naive and worst case is in Table A.2. To keep the hyperparameters in a more realistic frame, the parameters `OMP_NUM_TEAMS` and the scheduling algorithm are manually chosen. For the worst case, the variable `OMP_NUM_TEAMS` is set to 1024 because this is rather the default value for many applications. The results of 4.3.7 indicate that the coarse-grained scheduling algorithm does not work as intended. Therefore, the default scheduling algorithm is used for the worst case. For the worst case, no static tasks were used either. Other configurations, like the matrix size, are the same as in the base scenario. The measurements were conducted with an

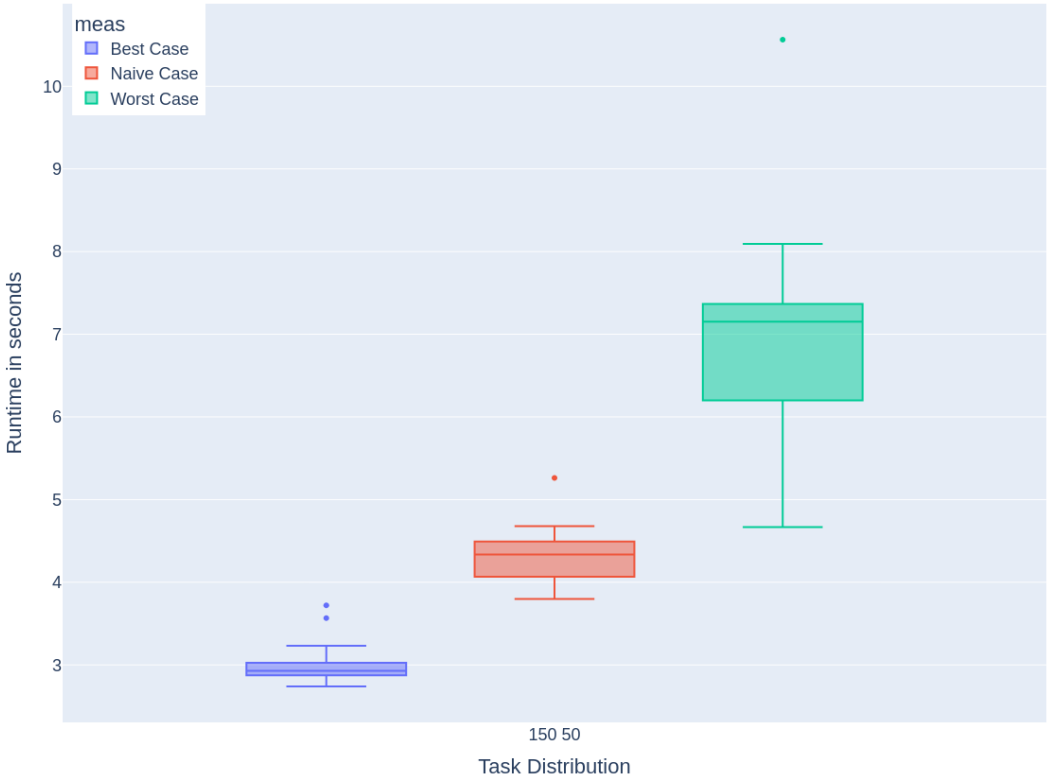


Figure 4.7.: The best case compared to the naive and the worst case. Lower is better.

unbalanced scenario since it tests the purpose of targetDART appropriately. For the measurements, the ANY device was used, since it had the best performance for the task size and imbalance of the base scenario.

Results: The results are plotted in Figure 4.7. The best case outperforms the other two cases with a speedup of 2.06× in comparison to the naive case and 3.4× in comparison to the worst case. The variance of the best case is very low, with a difference of 0.27 seconds between the upper and lower fences of the box plot. The difference between the upper and lower fence for the naive case is 0.88 seconds and for the worst case 3.42 seconds.

Discussion: The results show that tuning targetDART with the right hyperparameters can accelerate the execution a lot. Since none of the hyperparameters was able to achieve a speedup that was higher than about 1.36×, some of the benefits from the hyperparameters can be combined and stacked to higher overall speedups. This way, an overall speedup of 2.06× can be achieved. There are also differences between the base scenario and the naive case, but the difference between the naive case and the unpinned base scenario (measurement from 4.3.1) is about 300ms or

7.1%.

4.3.9. Guidelines for the hyperparameters

This section gives a short summary of the key findings for the hyperparameters and gives a guideline on what hyperparameters should be used.

Conclusion on benchmark 4.3.1: Using pinned memory provides a significant performance gain and should always be used, since it also does not change the performance of the CPU device. To determine why the ANY device has more overhead with unpinned memory than the OFFLOAD device needs more research.

Conclusion on benchmark 4.3.2: Placing the receiver thread and the scheduler thread resulted in the highest performance gain. It also produces stable results with low standard deviation. On the other hand, placing all threads on the same core resulted in the worst performance.

The benchmark did not test all possible configurations, but most of those are not promising. For example, placing the scheduler and receiver thread together and placing the executor threads on one core. This is similar to the best performing configuration so far, but the results of test cases 2 and 3 indicate that placing executor threads together decreases the performance. Other options would include placing threads on different NUMA nodes or on the other CPU socket, which could increase the performance if the application is memory-bound, but would rather decrease the performance because the threads would need to communicate across NUMA domains.

Conclusion on benchmark 3.1.2: NUMA node 3 performs best for Claix23 nodes and is recommended to use that node. For other compute nodes, the runtime threads should be located on the same CPU socket as the used GPUs.

The performance differences between the NUMA nodes are, in general, very small with a maximum of 5%. Therefore, optimizing that parameter is not as important as other parameters and might need specific tests for each distinct node that is used.

Conclusions on benchmark 4.3.4: The usage of multiple executor threads provides a high performance benefit. Using two executor threads already increases the performance by 27-31%. For using three executors, the performance gain is smaller, with about 2-5% and using one more thread increases performance by less than 1%. Therefore, using three executors is recommended, since using more executors than that barely improves the performance.

The behaviour of this parameter is strongly related to the hardware. It depends on the accelerator that is used, but it also depends on the CPU and the bandwidth of the PCI link. For other compute nodes, more executor threads might perform a lot better or might decrease the performance compared to fewer executors.

Conclusions on benchmark 4.3.5: Leaving `OMP_NUM_TEAMS` undefined achieves the best results. OpenMP internally handles the definition of that variable in that case. A common value for this parameter is 1024. For this value, the performance is very similar to the values 2048 and 4096. Leaving the parameter undefined improves the performance by 5-6%, which is also the recommendation for that parameter.

Conclusions on benchmark 4.3.6 The exact mechanisms for the overhead generation and for achieving performance benefits are unknown. Despite that, a low performance benefit was observed for 5 static tasks on each node, which equals 5% of the entire workload. Assigning 5% of the tasks statically to nodes is recommended, with the static tasks being equally distributed across all nodes, like in the benchmark.

Conclusions on benchmark 4.3.7: The results indicate that the coarse-grained algorithm does not work as intended, which also means that the combination of the coarse-grained and the fine-grained scheduling does not achieve good performance. Therefore, the fine-grained scheduling algorithm is recommended, since it always outperformed the coarse-grained algorithm in the unbalanced scenario by 89.5% for the ANY device or 6.3% for the CPU device and also outperformed the combination of both algorithms with 31.1% and 3.2%.

Based on the results of these benchmarks, the hyperparameters are separated into parameters that have a low impact on the performance with $< 5\%$ and parameters that have a high impact $\geq 5\%$.

Guidelines for hyperparameters: High impact($\geq 5\%$):

- Always use pinned memory. In the measurements, it improved the performance by 36.9% for the ANY device and 16.8% for the OFFLOAD device. For the CPU device, it barely increased the performance, but also did not decrease the performance. Therefore, using pinned memory is always recommended.
- Use multiple executor threads(3-4 threads). It decreased the runtime for the ANY device by about 36% and by 29% for the OFFLOAD device. To utilize this hyperparameter, pinned memory is required.
- Do not use coarse-grained scheduling. If coarse-grained scheduling was used, with or without the fine-grained scheduling combined, the performance dropped for unbalanced scenarios by at least 31.1% for the ANY device and by 3.2% for the CPU. Therefore, it is not recommended to use the coarse-grained scheduling strategy.
- Leave `OMP_NUM_TEAMS` undefined. Setting this hyperparameter to any of the tested values decreased the performance.
- Place scheduler and receiver thread together. This increased the performance by about 6% in comparison to placing all runtime threads on their own core.

Low impact($< 5\%$)

- Place the runtime threads on the same socket as the GPU. Placing the runtime threads on the same socket as the GPU achieved about 3-5% performance gain.
- Assign about 5% of the overall workload statically to different nodes. The static tasks should be evenly distributed to all nodes. This decreased the runtime by about 2.4% for the measurements.

Comparing a best case, a naive case and a realistic worst case for the hyperparameter selection, the best case outperformed the other cases. The speedup of the best case was $2.06\times$ compared to a naive case. This shows that tuning the hyperparameters can significantly speed up the application. The speedup compared to the worst case is $3.4\times$, so one can also decrease the performance significantly by setting the hyperparameters accordingly.

5. Conclusion and further Research

This thesis tested the possible use cases for which targetDART is useful and how to tune it with hyperparameters. Various benchmarks were conducted in order to derive guidelines for the use cases and hyperparameters.

For the use cases, targetDART generated for many situations small overheads of, for example, 400ms in 4.2.7 or 14-16% in 4.2.6, but often outperformed the references and scaled resiliently. Therefore, targetDART is useful for many scenarios and fulfills its purpose of accelerating the execution of imbalanced applications.

For the hyperparameters, the user can tune targetDART with its hyperparameters to half the execution time, as the comparison of the best and naive cases showed. But the worst case also shows that badly configured hyperparameters can decrease the performance in comparison to the naive case, so if no hyperparameters are manually tuned.

Further Research As pointed out at some benchmarks, like 4.2.7 and 3.1.2, more testing is required to understand the behaviour of targetDART. For example, the task size could potentially cover the overhead of the scheduling, like in Section 4.2.4. Further tests with scaling the task size for some benchmarks could reduce the overhead. The research about the task size could be used to refine the guidelines regarding task granularity.

Another topic for further investigation would be why the coarse-grained algorithm does not work as intended. The coarse-grained algorithm could be used to neutralize the worst case of the fine-grained scheduling algorithm. Furthermore, there are a lot of scheduling algorithms that could be used by targetDART. Other scheduling algorithms have the potential to speed up the balancing process, which currently induces overhead, although it is implemented in a non-blocking way.

Comparing targetDART to other tools could show how efficient targetDART is and could reveal how to implement a load balancing tool like targetDART with less overhead. It could also reveal that other tools are more suitable for certain scenarios. For example, if another tool like IRIS [17] or the work of Samfass et al. [24] would induce less overhead for CPU-only applications. The benchmark in 4.2.2 shows that the overhead of the targetDART CPU device is very high, so other tools could outperform targetDART for CPU-only applications.

A. Appendix for the Benchmarks

Listing A.1: The source code that was used to test the targetDART versions.

```
#include <stdlib.h>
#include <iostream>
#include <cstdlib>
#include <omp.h>
#include <mpi.h>
#include <string>

int main(int argc, char** argv) {

    int rank, size;
    int provided;
    int device;

    MPI_Init_thread(&argc, &argv, MPI_THREAD_MULTIPLE
       , &provided);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);

    // targetDART initialization
    //td_init((void *) &main);

    if (std::getenv("MODE") == NULL) {
        device = TARGETDART_ANY;
    } else {

        std::string mode = std::getenv("MODE");
    if (mode == "ANY_LOCAL") {
        device = TARGETDART_ANY_LOCAL;
    } else if (mode == "GPU_LOCAL") {
        device = TARGETDART_OFFLOAD_LOCAL;
    } else if (mode == "CPU") {
            device = TARGETDART_CPU;
        } else if (mode == "GPU") {
            device = TARGETDART_OFFLOAD;
        } else {
```

```
35              device = TARGETDART_ANY;
36          }
37      }
38
39      //std::cout << "MPI size, rank: " << size << ", "
            << rank << std::endl;
40
41      if (argc < 4 + size) {
42          std::cerr << "not enough arguments: For " <<
                size << " MPI processes you need at least
                " << 4 + size << " Arguments" << std::endl
                ;
43          exit(1);
44      }
45
46      int d1 = std::atoi(argv[1]);
47      int d2 = std::atoi(argv[2]);
48      int d3 = std::atoi(argv[3]);
49
50
51      int iter = std::atoi(argv[rank + 4]);
52
53      // enable pinned memory only when PINNED is
            explcitly enabled
54      bool pinned = false;
55      if (auto *pinned_env = std::getenv("PINNED")) {
56          std::string pinned_str = pinned_env;
57          pinned = !(pinned_str == "0" || pinned_str.
                empty());
58      }
59
60      double *A, *B, *C;
61
62      if (pinned) {
63          A = (double*)omp_alloc(d1 * d2 * sizeof(
                double), llvm_omp_target_host_mem_alloc);
64          B = (double*)omp_alloc(d2 * d3 * sizeof(
                double), llvm_omp_target_host_mem_alloc);
65          C = (double*)omp_alloc(iter * d1 * d3 *
                sizeof(double),
                llvm_omp_target_host_mem_alloc);
66      } else {
67          A = (double*) malloc(d1 * d2 * sizeof(double)
                );
```

```
68          B = (double*) malloc(d2 * d3 * sizeof(double)
                );
69          C = (double*) malloc(iter * d1 * d3 * sizeof(
                double));
70      }
71
72      for (int i = 0; i < d1 * d2; i++) {
73          A[i] = 1;
74      }
75      for (int i = 0; i < d2 * d3; i++) {
76          B[i] = 1;
77      }
78      for (int i = 0; i < iter * d1 * d3; i++) {
79          C[i] = 0;
80      }
81
82      MPI_Barrier(MPI_COMM_WORLD);
83      double time = omp_get_wtime();
84
85      for (int l = 0; l < iter; l++) {
86          double *C_l = C + l * d1 * d3;
87          #pragma omp target teams distribute parallel
                for map(from:C_l[0:d1*d3]) map(to:A[0:d1*
                d2]) map(to:B[0:d2*d3]) device(device)
                collapse(2) nowait
88          for (int i = 0; i < d1; i++) {
89              for (int k = 0; k < d3; k++) {
90                  C_l[i * d3 + k] = 0;
91                  for (int j = 0; j < d2; j++) {
92                      C_l[i * d3 + k] += A[i * d2 + j]
                            * B[j * d3 + k];
93                  }
94              }
95          }
96      }
97
98      #pragma omp taskwait
99
100     MPI_Barrier(MPI_COMM_WORLD);
101     time = omp_get_wtime() - time;
102
103     if (rank == 0) {
104         std::cout << "duration on process " << rank
                << ": " << time << std::endl;
```

```
105         //std::cout << "Result:  " << C[0] << std::
                endl;
106         int sum = 0;
107         for (int j = 0; j < d2; j++) {
108             sum += A[0 * d2 + j] * B[j * d3 + 0];
109         }
110         for (int i = 0; i < d1 * d3 * iter; i++) {
111             if (C[i] != sum) {
112                 std::cout << "Error: C[" << i << "] =
                        " << C[i] << " != " << sum << std
                       ::endl;
113                 break;
114             }
115         }
116     }
117
118     if (pinned) {
119         omp_free(A, llvm_omp_target_host_mem_alloc);
120         omp_free(B, llvm_omp_target_host_mem_alloc);
121         omp_free(C, llvm_omp_target_host_mem_alloc);
122     } else {
123         free(A);
124         free(B);
125         free(C);
126     }
127
128     //finalizeTargetDART();
129     MPI_Finalize();
130     return 0;
131 }
```

Listing A.2: The source code that was used to test the CPU reference.

```
1  #include <stdlib.h>
2  #include <iostream>
3  #include <cstdlib>
4  #include <omp.h>
5  #include <mpi.h>
6
7
8  int main(int argc, char** argv) {
9
10     int rank, size;
11     int provided;
12
```

```
13      MPI_Init_thread(&argc, &argv, MPI_THREAD_MULTIPLE
            , &provided);
14      MPI_Comm_rank(MPI_COMM_WORLD, &rank);
15      MPI_Comm_size(MPI_COMM_WORLD, &size);
16
17      if (argc < 4 + size) {
18          std::cerr << "not enough arguments: For " <<
                size << " MPI processes you need at least
                " << 4 +size << " Arguments" << std::endl;
19          exit(1);
20      }
21
22      int d1 = std::atoi(argv[1]);
23      int d2 = std::atoi(argv[2]);
24      int d3 = std::atoi(argv[3]);
25
26      int iter = std::atoi(argv[rank + 4]);
27
28      double* A = (double*) malloc(d1 * d2 * sizeof(
            double));
29      double* B = (double*) malloc(d2 * d3 * sizeof(
            double));
30      double* C = (double*) malloc(iter * d1 * d3 *
            sizeof(double));
31
32      for (int i = 0; i < d1 * d2; i++) {
33          A[i] = 1;
34      }
35      for (int i = 0; i < d2 * d3; i++) {
36          B[i] = 1;
37      }
38      for (int i = 0; i < iter * d1 * d3; i++) {
39          C[i] = 0;
40      }
41
42      MPI_Barrier(MPI_COMM_WORLD);
43      double time = omp_get_wtime();
44
45      for (int l = 0; l < iter; l++) {
46          double *C_l = C + l * d1 * d3;
47          #pragma omp parallel for collapse(2)
48          for (int i = 0; i < d1; i++) {
49              for (int k = 0; k < d3; k++) {
50                  C_l[i * d3 + k] = 0;
```

```
51                  for (int j = 0; j < d2; j++) {
52                      C_l[i * d3 + k] += A[i * d2 + j]
                            * B[j * d3 + k];
53                  }
54              }
55          }
56      }
57
58      #pragma omp taskwait
59
60      MPI_Barrier(MPI_COMM_WORLD);
61      time = omp_get_wtime() - time;
62
63      if (rank == 0) {
64          std::cout << "duration on process " << rank
                << ": " << time << std::endl;
65          //std::cout << "Result:  " << C[0] << std::
                endl;
66          int sum = 0;
67          for (int j = 0; j < d2; j++) {
68              sum += A[0 * d2 + j] * B[j * d3 + 0];
69          }
70          for (int i = 0; i < d1 * d3 * iter; i++) {
71              if (C[i] != sum) {
72                  std::cout << "Error: C[" << i << "] =
                        " << C[i] << " != " << sum << std
                        ::endl;
73                  break;
74              }
75          }
76      }
77
78      free(A);
79      free(B);
80      free(C);
81
82      MPI_Finalize();
83      return 0;
84  }
```

Listing A.3: The source code that was used to test the GPU reference.

```
1  #include <stdlib.h>
2  #include <stdint.h>
3  #include <iostream>
```

```
4  #include <cstdlib>
5  #include <omp.h>
6  #include <mpi.h>
7
8
9  int main(int argc, char** argv) {
10
11     int rank, size;
12     int provided;
13
14     MPI_Init_thread(&argc, &argv, MPI_THREAD_MULTIPLE
          , &provided);
15     MPI_Comm_rank(MPI_COMM_WORLD, &rank);
16     MPI_Comm_size(MPI_COMM_WORLD, &size);
17
18     if (argc < 4 + size) {
19         std::cerr << "not enough arguments: For " <<
              size << " MPI processes you need at least
              " << 4 +size << " Arguments" << std::endl;
20         exit(1);
21     }
22
23     int d1 = std::atoi(argv[1]);
24     int d2 = std::atoi(argv[2]);
25     int d3 = std::atoi(argv[3]);
26
27     int iter = std::atoi(argv[rank + 4]);
28
29     double * A = (double*) malloc(d1 * d2 * sizeof(
          double));
30     double * B = (double*) malloc(d2 * d3 * sizeof(
          double));
31     double * C = (double*) malloc(iter * d1 * d3 *
          sizeof(double));
32
33     for (int i = 0; i < d1 * d2; i++) {
34         A[i] = 1;
35     }
36     for (int i = 0; i < d2 * d3; i++) {
37         B[i] = 1;
38     }
39     for (int i = 0; i < d1 * d3; i++) {
40         C[i] = 0;
41     }
```

```
42
43      MPI_Barrier(MPI_COMM_WORLD);
44      double time = omp_get_wtime();
45
46      for (int l = 0; l < iter; l++) {
47          double *C_l = C + l * d1 * d3;
48          #pragma omp target teams distribute parallel
               for map(from:C_l[0:d1*d3]) map(to:A[0:d1*
               d2]) map(to:B[0:d2*d3]) map(to:d1,d2,d3)
               device(l%omp_get_num_devices()) collapse
               (2) nowait
49          for (int i = 0; i < d1; i++) {
50              for (int k = 0; k < d3; k++) {
51                  C_l[i * d3 + k] = 0;
52                  for (int j = 0; j < d2; j++) {
53                      C_l[i * d3 + k] += A[i * d2 + j]
                           * B[j * d3 + k];
54                  }
55              }
56          }
57      }
58
59      #pragma omp taskwait
60
61      MPI_Barrier(MPI_COMM_WORLD);
62      time = omp_get_wtime() - time;
63
64      if (rank == 0) {
65          std::cout << "duration on process " << rank
               << ": " << time << std::endl;
66          //std::cout << "Result:  " << C[0] << std::
               endl;
67          int sum = 0;
68          for (int j = 0; j < d2; j++) {
69              sum += A[0 * d2 + j] * B[j * d3 + 0];
70          }
71          for (int i = 0; i < d1 * d3 * iter; i++) {
72              if (C[i] != sum) {
73                  std::cout << "Error: C[" << i << "] =
                       " << C[i] << " != " << sum << std
                      ::endl;
74                  break;
75              }
76          }
```

```
77      }
78
79      free(A);
80      free(B);
81      free(C);
82
83      MPI_Finalize();
84      return 0;
85  }
```

Listing A.4: The source code of the script that generated the random load shifts. This function was executed 10 times with different seeds from 1 to 10.

```
1  def new_loads(seed, n=4):
2      res = []
3      limiter = 250
4      remain = 100 * n
5      random.seed(seed)
6      for i in range(n-1):
7          load = int(remain*random.random())
8          if load > limiter:
9              load = limiter
10         res.append(load)
11         remain-=load
12     res.append(remain)
13
14     # correctness check
15     sum = 0
16     for j in res:
17         sum += j
18     if sum != 100*n:
19         print('Error total load balance exceeds
               expected value')
20
21     return res
```

Listing A.5: The source code for heterogeneous task sizes for targetDART.

```
1  #include <stdlib.h>
2  #include <iostream>
3  #include <cstdlib>
4  #include <omp.h>
5  #include <mpi.h>
6  #include <string>
7
8  int main(int argc, char** argv) {
```

```
9
10      int rank, size;
11      int provided;
12      int device;
13
14      MPI_Init_thread(&argc, &argv, MPI_THREAD_MULTIPLE
            , &provided);
15      MPI_Comm_rank(MPI_COMM_WORLD, &rank);
16      MPI_Comm_size(MPI_COMM_WORLD, &size);
17
18      // targetDART initialization
19      //td_init((void *) &main);
20
21      if (std::getenv("MODE") == NULL) {
22          device = TARGETDART_ANY;
23      } else {
24
25          std::string mode = std::getenv("MODE");
26      if (mode == "ANY_LOCAL") {
27          device = TARGETDART_ANY_LOCAL;
28      } else if (mode == "GPU_LOCAL") {
29          device = TARGETDART_OFFLOAD_LOCAL;
30      } else if (mode == "CPU") {
31              device = TARGETDART_CPU;
32          } else if (mode == "GPU") {
33              device = TARGETDART_OFFLOAD;
34          } else {
35              device = TARGETDART_ANY;
36          }
37      }
38
39      //std::cout << "MPI size, rank: " << size << ", "
            << rank << std::endl;
40
41      if (argc < 4 + size) {
42          std::cerr << "not enough arguments: For " <<
                size << " MPI processes you need at least
                " << 4 + size << " Arguments" << std::endl
                ;
43          exit(1);
44      }
45
46      int d1 = std::atoi(argv[1]);
47      int total_load = 0;
```

```
48      for(int i = 4; i < argc; i++) {
49          total_load += std::atoi(argv[i + 4]);
50      }
51
52      int d2;
53      int d2_secondary;
54      if(rank == 0){
55          d2 = (int) std::atoi(argv[2]) * 1.5;
56          d2_secondary = (int) std::atoi(argv[2]) *
               0.5;
57      } else {
58          d2 = (int) std::atoi(argv[2]) * 0.5;
59          d2_secondary = (int) std::atoi(argv[2]) *
               1.5;
60      }
61      int d3 = std::atoi(argv[3]);
62
63
64      int iter = std::atoi(argv[rank + 4]);
65
66      // enable pinned memory only when PINNED is
           explcitly enabled
67      bool pinned = false;
68      if (auto *pinned_env = std::getenv("PINNED")) {
69          std::string pinned_str = pinned_env;
70          pinned = !(pinned_str == "0" || pinned_str.
               empty());
71      }
72
73      double *A, *B, *C;
74
75      if (pinned) {
76          A = (double*)omp_alloc(d1 * d2 * sizeof(
               double), llvm_omp_target_host_mem_alloc);
77          B = (double*)omp_alloc(d2 * d3 * sizeof(
               double), llvm_omp_target_host_mem_alloc);
78          C = (double*)omp_alloc(iter * d1 * d3 *
               sizeof(double),
               llvm_omp_target_host_mem_alloc);
79      } else {
80          A = (double*) malloc(d1 * d2 * sizeof(double)
               );
81          B = (double*) malloc(d2 * d3 * sizeof(double)
               );
```

```
82          C = (double*) malloc(iter * d1 * d3 * sizeof(
                double));
83      }
84
85      for (int i = 0; i < d1 * d2; i++) {
86          A[i] = 1;
87      }
88      for (int i = 0; i < d2 * d3; i++) {
89          B[i] = 1;
90      }
91      for (int i = 0; i < iter * d1 * d3; i++) {
92          C[i] = 0;
93      }
94
95      // just works for less than 200 tasks per node
96      int upper_bound = 100;
97      int rest = iter % upper_bound;
98      int firstloop = upper_bound * (int)(iter >= 100);
99
100     MPI_Barrier(MPI_COMM_WORLD);
101     double time = omp_get_wtime();
102
103     for (int l = 0; l < firstloop; l++) {
104         double *C_l = C + l * d1 * d3;
105         #pragma omp target teams distribute parallel
                for map(from:C_l[0:d1*d3]) map(to:A[0:d1*
                d2]) map(to:B[0:d2*d3]) device(device)
                collapse(2) nowait
106         for (int i = 0; i < d1; i++) {
107             for (int k = 0; k < d3; k++) {
108                 C_l[i * d3 + k] = 0;
109                 for (int j = 0; j < d2; j++) {
110                     C_l[i * d3 + k] += A[i * d2 + j]
                            * B[j * d3 + k];
111                 }
112             }
113         }
114     }
115     if(firstloop > 0) {
116         for (int l = 0; l < rest; l++) {
117             double *C_l = C + (l + firstloop) * d1 *
                    d3;
118             #pragma omp target teams distribute
                    parallel for map(from:C_l[0:d1*d3])
```

```
                map(to:A[0:d1*d2_secondary]) map(to:B
                [0:d2_secondary*d3]) device(device)
                collapse(2) nowait
119             for (int i = 0; i < d1; i++) {
120                 for (int k = 0; k < d3; k++) {
121                     C_l[i * d3 + k] = 0;
122                     for (int j = 0; j < d2_secondary;
                            j++) {
123                         C_l[i * d3 + k] += A[i *
                                d2_secondary + j] * B[j *
                                d3 + k];
124                     }
125                 }
126             }
127         }
128     } else {
129         for (int l = 0; l < rest; l++) {
130             double *C_l = C + l * d1 * d3;
131             #pragma omp target teams distribute
                    parallel for map(from:C_l[0:d1*d3])
                    map(to:A[0:d1*d2]) map(to:B[0:d2*d3])
                    device(device) collapse(2) nowait
132             for (int i = 0; i < d1; i++) {
133                 for (int k = 0; k < d3; k++) {
134                     C_l[i * d3 + k] = 0;
135                     for (int j = 0; j < d2; j++) {
136                         C_l[i * d3 + k] += A[i * d2 +
                                j] * B[j * d3 + k];
137                     }
138                 }
139             }
140         }
141     }
142
143     #pragma omp taskwait
144
145     MPI_Barrier(MPI_COMM_WORLD);
146     time = omp_get_wtime() - time;
147
148     if (rank == 0) {
149         std::cout << "duration on process " << rank
                << ": " << time << std::endl;
150         //std::cout << "Result:  " << C[0] << std::
                endl;
```

```
151         /*
152         int sum = 0;
153         for (int j = 0; j < d2; j++) {
154             sum += A[0 * d2 + j] * B[j * d3 + 0];
155         }
156         for (int j = 0; j < d2_secondary; j++) {
157             sum += A[0 * d2_secondary + j] * B[j * d3
                    + 0];
158         }
159         for (int i = 0; i < d1 * d3 * firstloop; i++)
                {
160             if (C[i] != sum) {
161                 std::cout << "Error: C[" << i << "] =
                        " << C[i] << " != " << sum << std
                        ::endl;
162                 //break;
163             }
164         }
165         // ... further checks
166         */
167     }
168
169     if (pinned) {
170         omp_free(A, llvm_omp_target_host_mem_alloc);
171         omp_free(B, llvm_omp_target_host_mem_alloc);
172         omp_free(C, llvm_omp_target_host_mem_alloc);
173     } else {
174         free(A);
175         free(B);
176         free(C);
177     }
178
179     //finalizeTargetDART();
180     MPI_Finalize();
181     return 0;
182 }
```

Listing A.6: The source code for heterogeneous task sizes for the CPU reference.

```
1 #include <stdlib.h>
2 #include <iostream>
3 #include <cstdlib>
4 #include <omp.h>
5 #include <mpi.h>
6
```

```
7
8  int main(int argc, char** argv) {
9
10     int rank, size;
11     int provided;
12
13     MPI_Init_thread(&argc, &argv, MPI_THREAD_MULTIPLE
          , &provided);
14     MPI_Comm_rank(MPI_COMM_WORLD, &rank);
15     MPI_Comm_size(MPI_COMM_WORLD, &size);
16
17     if (argc < 4 + size) {
18         std::cerr << "not enough arguments: For " <<
              size << " MPI processes you need at least
              " << 4 +size << " Arguments" << std::endl;
19         exit(1);
20     }
21
22     int d1 = std::atoi(argv[1]);
23     int total_load = 0;
24     for(int i = 4; i < argc; i++) {
25         total_load += std::atoi(argv[i + 4]);
26     }
27
28     int d2;
29     int d2_secondary;
30     if(rank == 0){
31         d2 = (int) std::atoi(argv[2]) * 1.5;
32         d2_secondary = (int) std::atoi(argv[2]) *
              0.5;
33     } else {
34         d2 = (int) std::atoi(argv[2]) * 0.5;
35         d2_secondary = (int) std::atoi(argv[2]) *
              1.5;
36     }
37     int d3 = std::atoi(argv[3]);
38
39     int iter = std::atoi(argv[rank + 4]);
40
41     double* A = (double*) malloc(d1 * d2 * sizeof(
          double));
42     double* B = (double*) malloc(d2 * d3 * sizeof(
          double));
43     double* C = (double*) malloc(iter * d1 * d3 *
```

```
        sizeof(double));
44
45      for (int i = 0; i < d1 * d2; i++) {
46          A[i] = 1;
47      }
48      for (int i = 0; i < d2 * d3; i++) {
49          B[i] = 1;
50      }
51      for (int i = 0; i < iter * d1 * d3; i++) {
52          C[i] = 0;
53      }
54
55      // just works for less than 200 tasks per node
56      int upper_bound = 100;
57      int rest = iter % upper_bound;
58      int firstloop = upper_bound * (int)(iter >= 100);
59
60      MPI_Barrier(MPI_COMM_WORLD);
61      double time = omp_get_wtime();
62
63      for (int l = 0; l < firstloop; l++) {
64          double *C_l = C + l * d1 * d3;
65          #pragma omp parallel for collapse(2)
66          for (int i = 0; i < d1; i++) {
67              for (int k = 0; k < d3; k++) {
68                  C_l[i * d3 + k] = 0;
69                  for (int j = 0; j < d2; j++) {
70                      C_l[i * d3 + k] += A[i * d2 + j]
                            * B[j * d3 + k];
71                  }
72              }
73          }
74      }
75      if(firstloop > 0) {
76          for (int l = 0; l < rest; l++) {
77              double *C_l = C + (l + firstloop) * d1 *
                    d3;
78              #pragma omp parallel for collapse(2)
79              for (int i = 0; i < d1; i++) {
80                  for (int k = 0; k < d3; k++) {
81                      C_l[i * d3 + k] = 0;
82                      for (int j = 0; j < d2_secondary;
                            j++) {
83                          C_l[i * d3 + k] += A[i *
```

```
                            d2_secondary + j] * B[j *
                            d3 + k];
84                      }
85                  }
86              }
87          }
88      } else {
89          for (int l = 0; l < firstloop; l++) {
90              double *C_l = C + l * d1 * d3;
91              #pragma omp parallel for collapse(2)
92              for (int i = 0; i < d1; i++) {
93                  for (int k = 0; k < d3; k++) {
94                      C_l[i * d3 + k] = 0;
95                      for (int j = 0; j < d2; j++) {
96                          C_l[i * d3 + k] += A[i * d2 +
                                j] * B[j * d3 + k];
97                      }
98                  }
99              }
100         }
101     }
102
103     #pragma omp taskwait
104
105     MPI_Barrier(MPI_COMM_WORLD);
106     time = omp_get_wtime() - time;
107
108     if (rank == 0) {
109         std::cout << "duration on process " << rank
               << ": " << time << std::endl;
110         //std::cout << "Result:  " << C[0] << std::
               endl;
111         /*
112         int sum = 0;
113         for (int j = 0; j < d2; j++) {
114             sum += A[0 * d2 + j] * B[j * d3 + 0];
115         }
116         for (int i = 0; i < d1 * d3 * iter; i++) {
117             if (C[i] != sum) {
118                 std::cout << "Error: C[" << i << "] =
                        " << C[i] << " != " << sum << std
                       ::endl;
119                 break;
120             }
```

```
121             }
122             */
123         }
124
125         free(A);
126         free(B);
127         free(C);
128
129         MPI_Finalize();
130         return 0;
131     }
```

Listing A.7: The source code for heterogeneous task sizes for the GPU reference.

```
1   #include <stdlib.h>
2   #include <stdint.h>
3   #include <iostream>
4   #include <cstdlib>
5   #include <omp.h>
6   #include <mpi.h>
7
8
9   int main(int argc, char** argv) {
10
11      int rank, size;
12      int provided;
13
14      MPI_Init_thread(&argc, &argv, MPI_THREAD_MULTIPLE
            , &provided);
15      MPI_Comm_rank(MPI_COMM_WORLD, &rank);
16      MPI_Comm_size(MPI_COMM_WORLD, &size);
17
18      if (argc < 4 + size) {
19          std::cerr << "not enough arguments: For " <<
                size << " MPI processes you need at least
                " << 4 +size << " Arguments" << std::endl;
20          exit(1);
21      }
22
23      int d1 = std::atoi(argv[1]);
24      int total_load = 0;
25      for(int i = 4; i < argc; i++) {
26          total_load += std::atoi(argv[i + 4]);
27      }
28
```

```
29      int d2;
30      int d2_secondary;
31      if(rank == 0){
32          d2 = (int) std::atoi(argv[2]) * 1.5;
33          d2_secondary = (int) std::atoi(argv[2]) *
                0.5;
34      } else {
35          d2 = (int) std::atoi(argv[2]) * 0.5;
36          d2_secondary = (int) std::atoi(argv[2]) *
                1.5;
37      }
38      int d3 = std::atoi(argv[3]);
39
40      int iter = std::atoi(argv[rank + 4]);
41
42      double * A = (double*) malloc(d1 * d2 * sizeof(
            double));
43      double * B = (double*) malloc(d2 * d3 * sizeof(
            double));
44      double * C = (double*) malloc(iter * d1 * d3 *
            sizeof(double));
45
46      for (int i = 0; i < d1 * d2; i++) {
47          A[i] = 1;
48      }
49      for (int i = 0; i < d2 * d3; i++) {
50          B[i] = 1;
51      }
52      for (int i = 0; i < d1 * d3; i++) {
53          C[i] = 0;
54      }
55
56      // just works for less than 200 tasks per node
57      int upper_bound = 100;
58      int rest = iter % upper_bound;
59      int firstloop = upper_bound * (int)(iter >= 100);
60
61      MPI_Barrier(MPI_COMM_WORLD);
62      double time = omp_get_wtime();
63
64      for (int l = 0; l < firstloop; l++) {
65          double *C_l = C + l * d1 * d3;
66          #pragma omp target teams distribute parallel
                for map(from:C_l[0:d1*d3]) map(to:A[0:d1*
```

```
            d2]) map(to:B[0:d2*d3]) map(to:d1,d2,d3)
            device(l%omp_get_num_devices()) collapse
            (2) nowait
67          for (int i = 0; i < d1; i++) {
68              for (int k = 0; k < d3; k++) {
69                  C_l[i * d3 + k] = 0;
70                  for (int j = 0; j < d2; j++) {
71                      C_l[i * d3 + k] += A[i * d2 + j]
                            * B[j * d3 + k];
72                  }
73              }
74          }
75      }
76      if(firstloop > 0) {
77          for (int l = 0; l < rest; l++) {
78              double *C_l = C + (l + firstloop) * d1 *
                    d3;
79              #pragma omp target teams distribute
                    parallel for map(from:C_l[0:d1*d3])
                    map(to:A[0:d1*d2]) map(to:B[0:d2*d3])
                    map(to:d1,d2,d3) device(l%
                    omp_get_num_devices()) collapse(2)
                    nowait
80              for (int i = 0; i < d1; i++) {
81                  for (int k = 0; k < d3; k++) {
82                      C_l[i * d3 + k] = 0;
83                      for (int j = 0; j < d2_secondary;
                            j++) {
84                          C_l[i * d3 + k] += A[i *
                                d2_secondary + j] * B[j *
                                d3 + k];
85                      }
86                  }
87              }
88          }
89      } else {
90          for (int l = 0; l < firstloop; l++) {
91              double *C_l = C + l * d1 * d3;
92              #pragma omp target teams distribute
                    parallel for map(from:C_l[0:d1*d3])
                    map(to:A[0:d1*d2]) map(to:B[0:d2*d3])
                    map(to:d1,d2,d3) device(l%
                    omp_get_num_devices()) collapse(2)
                    nowait
```

```
93              for (int i = 0; i < d1; i++) {
94                  for (int k = 0; k < d3; k++) {
95                      C_l[i * d3 + k] = 0;
96                      for (int j = 0; j < d2; j++) {
97                          C_l[i * d3 + k] += A[i * d2 +
                                j] * B[j * d3 + k];
98                      }
99                  }
100             }
101         }
102     }
103
104     #pragma omp taskwait
105
106     MPI_Barrier(MPI_COMM_WORLD);
107     time = omp_get_wtime() - time;
108
109     if (rank == 0) {
110         std::cout << "duration on process " << rank
               << ": " << time << std::endl;
111         //std::cout << "Result:  " << C[0] << std::
               endl;
112         /*
113         int sum = 0;
114         for (int j = 0; j < d2; j++) {
115             sum += A[0 * d2 + j] * B[j * d3 + 0];
116         }
117         for (int i = 0; i < d1 * d3 * iter; i++) {
118             if (C[i] != sum) {
119                 std::cout << "Error: C[" << i << "] =
                        " << C[i] << " != " << sum << std
                       ::endl;
120                 break;
121             }
122         }
123         */
124     }
125
126     free(A);
127     free(B);
128     free(C);
129
130     MPI_Finalize();
131     return 0;
```

```
132 }
```

Listing A.8: The source code for delayed task generation. This code generates $\frac{1}{2}$ of the at the beginning. The other half after the time of the last parameter in milliseconds.

```
1  #include <stdlib.h>
2  #include <iostream>
3  #include <cstdlib>
4  #include <omp.h>
5  #include <mpi.h>
6  #include <string>
7
8  #include <chrono>
9  #include <thread>
10
11
12 int main(int argc, char** argv) {
13
14     int rank, size;
15     int provided;
16     int device;
17
18     MPI_Init_thread(&argc, &argv, MPI_THREAD_MULTIPLE
       , &provided);
19     MPI_Comm_rank(MPI_COMM_WORLD, &rank);
20     MPI_Comm_size(MPI_COMM_WORLD, &size);
21
22     // targetDART initialization
23     //td_init((void *) &main);
24
25     if (std::getenv("MODE") == NULL) {
26         device = TARGETDART_ANY;
27     } else {
28
29         std::string mode = std::getenv("MODE");
30     if (mode == "ANY_LOCAL") {
31         device = TARGETDART_ANY_LOCAL;
32     } else if (mode == "GPU_LOCAL") {
33         device = TARGETDART_OFFLOAD_LOCAL;
34     } else if (mode == "CPU") {
35             device = TARGETDART_CPU;
36         } else if (mode == "GPU") {
37             device = TARGETDART_OFFLOAD;
38         } else {
```

```
39              device = TARGETDART_ANY;
40          }
41      }
42
43      //std::cout << "MPI size, rank: " << size << ", "
            << rank << std::endl;
44
45      if (argc < 4 + size) {
46          std::cerr << "not enough arguments: For " <<
                size << " MPI processes you need at least
                " << 4 + size << " Arguments" << std::endl
                ;
47          exit(1);
48      }
49
50      int d1 = std::atoi(argv[1]);
51      int d2 = std::atoi(argv[2]);
52      int d3 = std::atoi(argv[3]);
53
54
55      int iter = std::atoi(argv[rank + 4]);
56
57      int delay = std::atoi(argv[argc-1]); // if
            enabled there will be one task at the end
            which will double the tasknumbers, else every
            task will generate a new task
58
59      // enable pinned memory only when PINNED is
            explcitly enabled
60      bool pinned = false;
61      if (auto *pinned_env = std::getenv("PINNED")) {
62          std::string pinned_str = pinned_env;
63          pinned = !(pinned_str == "0" || pinned_str.
                empty());
64      }
65
66      double *A, *B, *C;
67
68      if (pinned) {
69          A = (double*)omp_alloc(d1 * d2 * sizeof(
                double), llvm_omp_target_host_mem_alloc);
70          B = (double*)omp_alloc(d2 * d3 * sizeof(
                double), llvm_omp_target_host_mem_alloc);
71          C = (double*)omp_alloc(2 * iter * d1 * d3 *
```

```
            sizeof(double),
            llvm_omp_target_host_mem_alloc);
72      } else {
73          A = (double*) malloc(d1 * d2 * sizeof(double)
                );
74          B = (double*) malloc(d2 * d3 * sizeof(double)
                );
75          C = (double*) malloc(2 * iter * d1 * d3 *
                sizeof(double));
76      }
77
78      for (int i = 0; i < d1 * d2; i++) {
79          A[i] = 1;
80      }
81      for (int i = 0; i < d2 * d3; i++) {
82          B[i] = 1;
83      }
84      for (int i = 0; i < 2 * iter * d1 * d3; i++) {
85          C[i] = 0;
86      }
87
88      MPI_Barrier(MPI_COMM_WORLD);
89      double time = omp_get_wtime();
90
91      for (int l = 0; l < iter; l++) {
92          double *C_l = C + l * d1 * d3;
93          #pragma omp target teams distribute parallel
              for map(from:C_l[0:d1*d3]) map(to:A[0:d1*
              d2]) map(to:B[0:d2*d3]) device(device)
              collapse(2) nowait
94          for (int i = 0; i < d1; i++) {
95              for (int k = 0; k < d3; k++) {
96                  C_l[i * d3 + k] = 0;
97                  for (int j = 0; j < d2; j++) {
98                      C_l[i * d3 + k] += A[i * d2 + j]
                            * B[j * d3 + k];
99                  }
100             }
101         }
102     }
103
104     std::this_thread::sleep_for(std::chrono::
            milliseconds(delay));
105
```

```
106     for (int l = 0; l < iter; l++) {
107         double *C_f = C + (l + iter) * d1 * d3;
108         #pragma omp target teams distribute parallel
               for map(from:C_f[0:d1*d3]) map(to:A[0:d1*
               d2]) map(to:B[0:d2*d3]) device(device)
               collapse(2) nowait
109         for (int i = 0; i < d1; i++) {
110             for (int k = 0; k < d3; k++) {
111                 C_f[i * d3 + k] = 0;
112                 for (int j = 0; j < d2; j++) {
113                     C_f[i * d3 + k] += A[i * d2 + j]
                           * B[j * d3 + k];
114                 }
115             }
116         }
117     }
118
119     #pragma omp taskwait
120
121     MPI_Barrier(MPI_COMM_WORLD);
122     time = omp_get_wtime() - time;
123
124     if (rank == 0) {
125         std::cout << "duration on process " << rank
               << ": " << time << std::endl;
126         //std::cout << "Result:  " << C[0] << std::
               endl;
127         int sum = 0;
128         for (int j = 0; j < d2; j++) {
129             sum += A[0 * d2 + j] * B[j * d3 + 0];
130         }
131         for (int i = 0; i < d1 * d3 * iter; i++) {
132             if (C[i] != sum) {
133                 std::cout << "Error: C[" << i << "] =
                       " << C[i] << " != " << sum << std
                       ::endl;
134                 break;
135             }
136         }
137     }
138
139     if (pinned) {
140         omp_free(A, llvm_omp_target_host_mem_alloc);
141         omp_free(B, llvm_omp_target_host_mem_alloc);
```

```
142             omp_free(C, llvm_omp_target_host_mem_alloc);
143         } else {
144             free(A);
145             free(B);
146             free(C);
147         }
148 
149         //finalizeTargetDART();
150         MPI_Finalize();
151         return 0;
152 }
```

Listing A.9: The source code for statically assigned tasks according to benchmark 4.3.6. This version generates the static tasks first

```
1  #include <stdlib.h>
2  #include <iostream>
3  #include <cstdlib>
4  #include <omp.h>
5  #include <mpi.h>
6  #include <string>
7  
8  int main(int argc, char** argv) {
9  
10     int rank, size;
11     int provided;
12     int device;
13     int device_static;
14 
15     MPI_Init_thread(&argc, &argv, MPI_THREAD_MULTIPLE
          , &provided);
16     MPI_Comm_rank(MPI_COMM_WORLD, &rank);
17     MPI_Comm_size(MPI_COMM_WORLD, &size);
18 
19     // targetDART initialization
20     //td_init((void *) &main);
21 
22     if (std::getenv("MODE") == NULL) {
23         device = TARGETDART_ANY;
24         device_static = TARGETDART_ANY_LOCAL;
25     } else {
26         std::string mode = std::getenv("MODE");
27        if (mode == "CPU") {
28             device = TARGETDART_CPU;
29             device_static = TARGETDART_CPU_LOCAL;
```

```
30          } else if (mode == "GPU") {
31              device = TARGETDART_OFFLOAD;
32              device_static = TARGETDART_OFFLOAD_LOCAL;
33          } else {
34              device = TARGETDART_ANY;
35              device_static = TARGETDART_ANY_LOCAL;
36          }
37      }
38
39      //std::cout << "MPI size, rank: " << size << ", "
            << rank << std::endl;
40
41      if (argc < 4 + size) {
42          std::cerr << "not enough arguments: For " <<
                size << " MPI processes you need at least
                " << 4 + size << " Arguments" << std::endl
                ;
43          exit(1);
44      }
45
46      int d1 = std::atoi(argv[1]);
47      int d2 = std::atoi(argv[2]);
48      int d3 = std::atoi(argv[3]);
49
50      int st = std::atoi(argv[2*rank + 4]);
51      int dy = std::atoi(argv[2*rank + 4 + 1]);
52      int iter = st + dy;
53
54      // enable pinned memory only when PINNED is
            explcitly enabled
55      bool pinned = false;
56      if (auto *pinned_env = std::getenv("PINNED")) {
57          std::string pinned_str = pinned_env;
58          pinned = !(pinned_str == "0" || pinned_str.
                empty());
59      }
60
61      double *A, *B, *C;
62
63      if (pinned) {
64          A = (double*)omp_alloc(d1 * d2 * sizeof(
                double), llvm_omp_target_host_mem_alloc);
65          B = (double*)omp_alloc(d2 * d3 * sizeof(
                double), llvm_omp_target_host_mem_alloc);
```

```
66          C = (double*)omp_alloc(iter * d1 * d3 *
                sizeof(double),
                llvm_omp_target_host_mem_alloc);
67      } else {
68          A = (double*) malloc(d1 * d2 * sizeof(double)
                );
69          B = (double*) malloc(d2 * d3 * sizeof(double)
                );
70          C = (double*) malloc(iter * d1 * d3 * sizeof(
                double));
71      }
72
73      for (int i = 0; i < d1 * d2; i++) {
74          A[i] = 1;
75      }
76      for (int i = 0; i < d2 * d3; i++) {
77          B[i] = 1;
78      }
79      for (int i = 0; i < iter * d1 * d3; i++) {
80          C[i] = 0;
81      }
82
83      MPI_Barrier(MPI_COMM_WORLD);
84      double time = omp_get_wtime();
85
86      for (int l = 0; l < st; l++) {
87          double *C_l = C + l * d1 * d3;
88          #pragma omp target teams distribute parallel
                for map(from:C_l[0:d1*d3]) map(to:A[0:d1*
                d2]) map(to:B[0:d2*d3]) device(
                device_static) collapse(2) nowait
89          for (int i = 0; i < d1; i++) {
90              for (int k = 0; k < d3; k++) {
91                  C_l[i * d3 + k] = 0;
92                  for (int j = 0; j < d2; j++) {
93                      C_l[i * d3 + k] += A[i * d2 + j]
                            * B[j * d3 + k];
94                  }
95              }
96          }
97      }
98      for (int l = 0; l < dy; l++) {
99          double *C_l = C + (st + l) * d1 * d3;
100         #pragma omp target teams distribute parallel
```

```
            for map(from:C_l[0:d1*d3]) map(to:A[0:d1*
            d2]) map(to:B[0:d2*d3]) device(device)
            collapse(2) nowait
101         for (int i = 0; i < d1; i++) {
102             for (int k = 0; k < d3; k++) {
103                 C_l[i * d3 + k] = 0;
104                 for (int j = 0; j < d2; j++) {
105                     C_l[i * d3 + k] += A[i * d2 + j]
                            * B[j * d3 + k];
106                 }
107             }
108         }
109     }
110
111     #pragma omp taskwait
112
113     MPI_Barrier(MPI_COMM_WORLD);
114     time = omp_get_wtime() - time;
115
116     if (rank == 0) {
117         std::cout << "duration on process " << rank
               << ": " << time << std::endl;
118         //std::cout << "Result:  " << C[0] << std::
               endl;
119         int sum = 0;
120         for (int j = 0; j < d2; j++) {
121             sum += A[0 * d2 + j] * B[j * d3 + 0];
122         }
123         for (int i = 0; i < d1 * d3 * iter; i++) {
124             if (C[i] != sum) {
125                 std::cout << "Error: C[" << i << "] =
                        " << C[i] << " != " << sum << std
                       ::endl;
126                 break;
127             }
128         }
129     }
130
131     if (pinned) {
132         omp_free(A, llvm_omp_target_host_mem_alloc);
133         omp_free(B, llvm_omp_target_host_mem_alloc);
134         omp_free(C, llvm_omp_target_host_mem_alloc);
135     } else {
136         free(A);
```

```
137             free(B);
138             free(C);
139         }
140
141         //finalizeTargetDART();
142         MPI_Finalize();
143         return 0;
144     }
```

Listing A.10: The source code for statically assigned tasks according to benchmark 4.3.6. This version generates the static tasks secondly

```
1   #include <stdlib.h>
2   #include <iostream>
3   #include <cstdlib>
4   #include <omp.h>
5   #include <mpi.h>
6   #include <string>
7
8   int main(int argc, char** argv) {
9
10      int rank, size;
11      int provided;
12      int device;
13      int device_static;
14
15      MPI_Init_thread(&argc, &argv, MPI_THREAD_MULTIPLE
            , &provided);
16      MPI_Comm_rank(MPI_COMM_WORLD, &rank);
17      MPI_Comm_size(MPI_COMM_WORLD, &size);
18
19      // targetDART initialization
20      //td_init((void *) &main);
21
22      if (std::getenv("MODE") == NULL) {
23          device = TARGETDART_ANY;
24          device_static = TARGETDART_ANY_LOCAL;
25      } else {
26          std::string mode = std::getenv("MODE");
27         if (mode == "CPU") {
28              device = TARGETDART_CPU;
29              device_static = TARGETDART_CPU_LOCAL;
30          } else if (mode == "GPU") {
31              device = TARGETDART_OFFLOAD;
32              device_static = TARGETDART_OFFLOAD_LOCAL;
```

```
33          } else {
34              device = TARGETDART_ANY;
35              device_static = TARGETDART_ANY_LOCAL;
36          }
37      }
38
39      //std::cout << "MPI size, rank: " << size << ", "
            << rank << std::endl;
40
41      if (argc < 4 + size) {
42          std::cerr << "not enough arguments: For " <<
                size << " MPI processes you need at least
                " << 4 + size << " Arguments" << std::endl
                ;
43          exit(1);
44      }
45
46      int d1 = std::atoi(argv[1]);
47      int d2 = std::atoi(argv[2]);
48      int d3 = std::atoi(argv[3]);
49
50      int st = std::atoi(argv[2*rank + 4]);
51      int dy = std::atoi(argv[2*rank + 4 + 1]);
52      int iter = st + dy;
53
54      // enable pinned memory only when PINNED is
            explcitly enabled
55      bool pinned = false;
56      if (auto *pinned_env = std::getenv("PINNED")) {
57          std::string pinned_str = pinned_env;
58          pinned = !(pinned_str == "0" || pinned_str.
                empty());
59      }
60
61      double *A, *B, *C;
62
63      if (pinned) {
64          A = (double*)omp_alloc(d1 * d2 * sizeof(
                double), llvm_omp_target_host_mem_alloc);
65          B = (double*)omp_alloc(d2 * d3 * sizeof(
                double), llvm_omp_target_host_mem_alloc);
66          C = (double*)omp_alloc(iter * d1 * d3 *
                sizeof(double),
                llvm_omp_target_host_mem_alloc);
```

```
67      } else {
68          A = (double*) malloc(d1 * d2 * sizeof(double)
               );
69          B = (double*) malloc(d2 * d3 * sizeof(double)
               );
70          C = (double*) malloc(iter * d1 * d3 * sizeof(
               double));
71      }
72
73      for (int i = 0; i < d1 * d2; i++) {
74          A[i] = 1;
75      }
76      for (int i = 0; i < d2 * d3; i++) {
77          B[i] = 1;
78      }
79      for (int i = 0; i < iter * d1 * d3; i++) {
80          C[i] = 0;
81      }
82
83      MPI_Barrier(MPI_COMM_WORLD);
84      double time = omp_get_wtime();
85      for (int l = 0; l < dy; l++) {
86          double *C_l = C + (st + l) * d1 * d3;
87          #pragma omp target teams distribute parallel
               for map(from:C_l[0:d1*d3]) map(to:A[0:d1*
               d2]) map(to:B[0:d2*d3]) device(device)
               collapse(2) nowait
88          for (int i = 0; i < d1; i++) {
89              for (int k = 0; k < d3; k++) {
90                  C_l[i * d3 + k] = 0;
91                  for (int j = 0; j < d2; j++) {
92                      C_l[i * d3 + k] += A[i * d2 + j]
                           * B[j * d3 + k];
93                  }
94              }
95          }
96      }
97
98      for (int l = 0; l < st; l++) {
99          double *C_l = C + l * d1 * d3;
100         #pragma omp target teams distribute parallel
               for map(from:C_l[0:d1*d3]) map(to:A[0:d1*
               d2]) map(to:B[0:d2*d3]) device(
               device_static) collapse(2) nowait
```

```
101         for (int i = 0; i < d1; i++) {
102             for (int k = 0; k < d3; k++) {
103                 C_l[i * d3 + k] = 0;
104                 for (int j = 0; j < d2; j++) {
105                     C_l[i * d3 + k] += A[i * d2 + j]
                            * B[j * d3 + k];
106                 }
107             }
108         }
109     }
110
111     #pragma omp taskwait
112
113     MPI_Barrier(MPI_COMM_WORLD);
114     time = omp_get_wtime() - time;
115
116     if (rank == 0) {
117         std::cout << "duration on process " << rank
               << ": " << time << std::endl;
118         //std::cout << "Result:  " << C[0] << std::
               endl;
119         int sum = 0;
120         for (int j = 0; j < d2; j++) {
121             sum += A[0 * d2 + j] * B[j * d3 + 0];
122         }
123         for (int i = 0; i < d1 * d3 * iter; i++) {
124             if (C[i] != sum) {
125                 std::cout << "Error: C[" << i << "] =
                        " << C[i] << " != " << sum << std
                       ::endl;
126                 break;
127             }
128         }
129     }
130
131     if (pinned) {
132         omp_free(A, llvm_omp_target_host_mem_alloc);
133         omp_free(B, llvm_omp_target_host_mem_alloc);
134         omp_free(C, llvm_omp_target_host_mem_alloc);
135     } else {
136         free(A);
137         free(B);
138         free(C);
139     }
```

```
140
141     //finalizeTargetDART();
142     MPI_Finalize();
143     return 0;
144 }
```

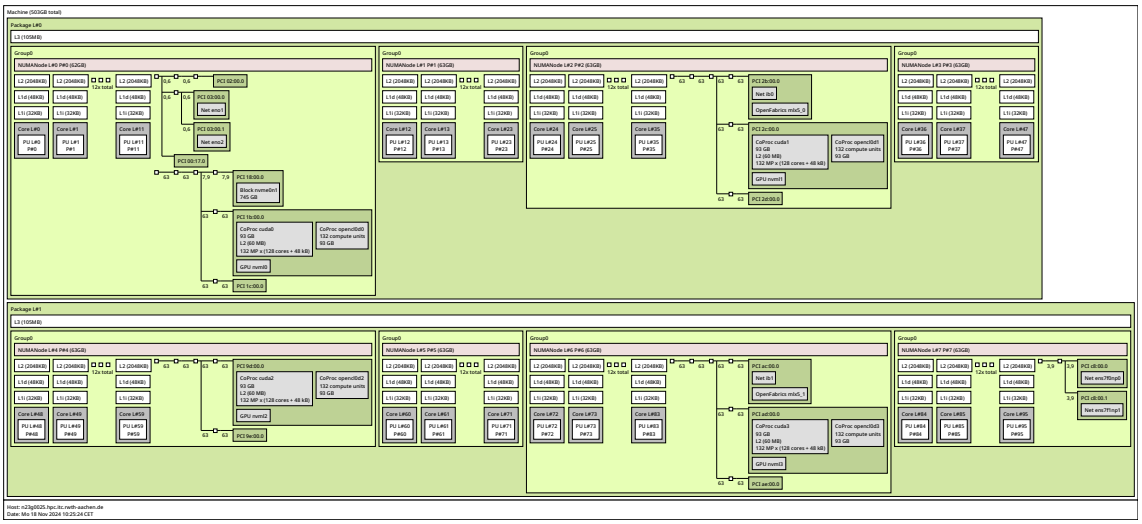


Figure A.1.: The topology of a single GPU node of Claix23.

Testcase Nr.	Task distribution	median runtime	standard deviation
0	100 100 100 100	17.79	0.27
1	53 250 74 23	28.24	0.47
2	95 165 51 89	35.78	0.56
3	95 165 51 89	23.44	0.28
4	94 31 108 167	11.24	0.31
5	249 112 31 8	33.16	0.5
6	250 123 13 14	33.85	0.71
7	129 40 150 81	15.65	0.34
8	90 250 7 53	30.52	0.43
9	185 80 18 117	23.97	0.3

Table A.1.: A table of the results of the CPU version for the random load imbalance benchmark. Lower is better.

Hyperparameters	Best case	Naive case	Worst case
`OMP_NUM_THREADS`	96	96	96
Pinned memory	yes	no	no
`TD_EXECUTORS_PER_DEVICE`	3	1	1
`TD_MANAGEMENT`	24,24,25,26,27,28,29	0,1,2,3	84,84,84,84
`OMP_NUM_TEAMS`	None	None	1024
Static load	5 tasks per node	no	no
Scheduling algorithm	default	default	default

Table A.2.: Hyperparameters for the best, naive and worst case.

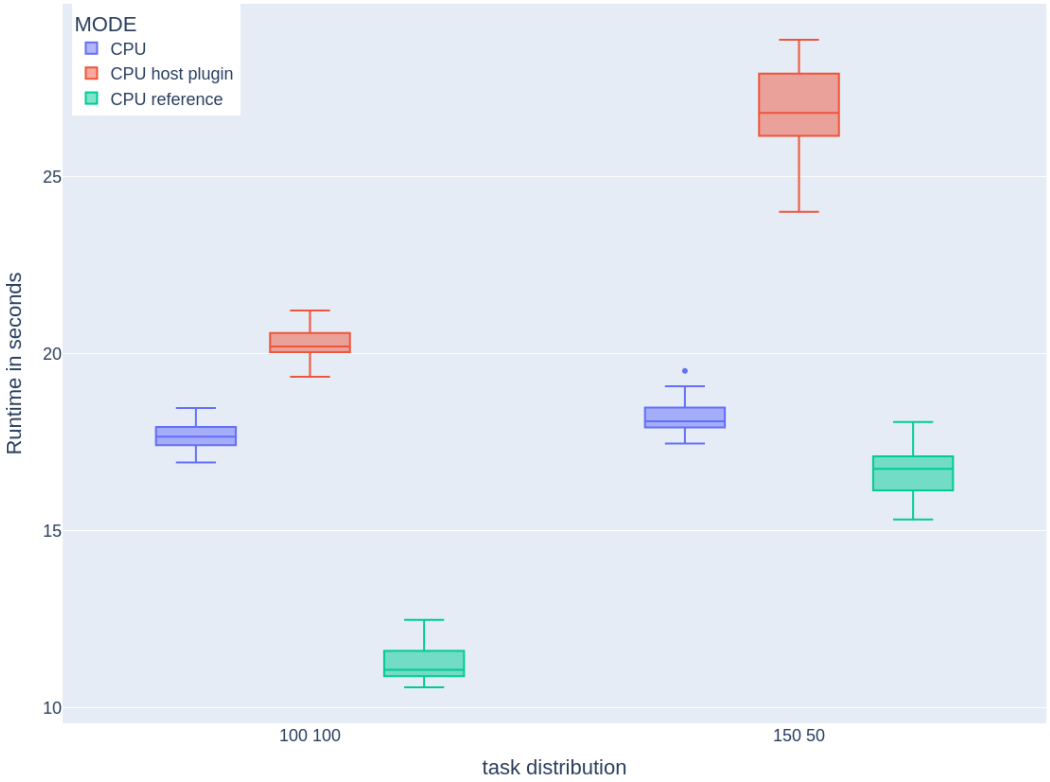


Figure A.2.: Results of the Benchmark 4.2.2. Lower is better.

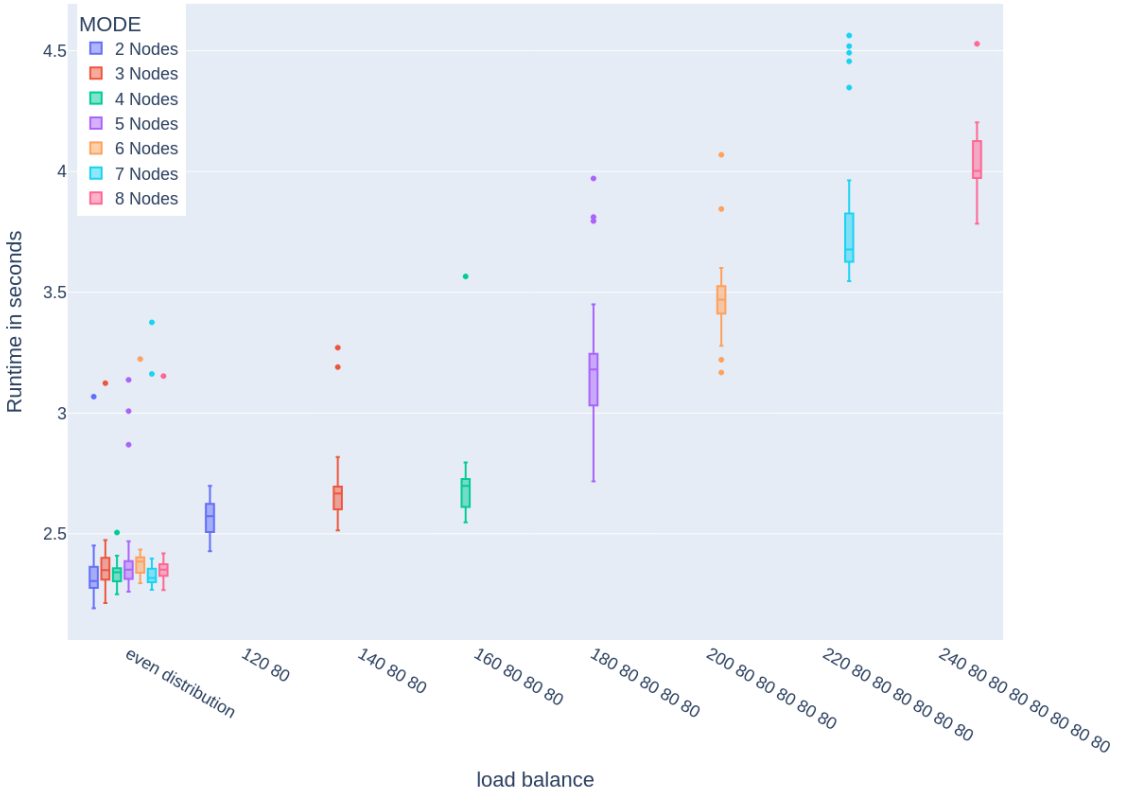


Figure A.3.: Results of the Benchmark 4.2.4. Lower is better.

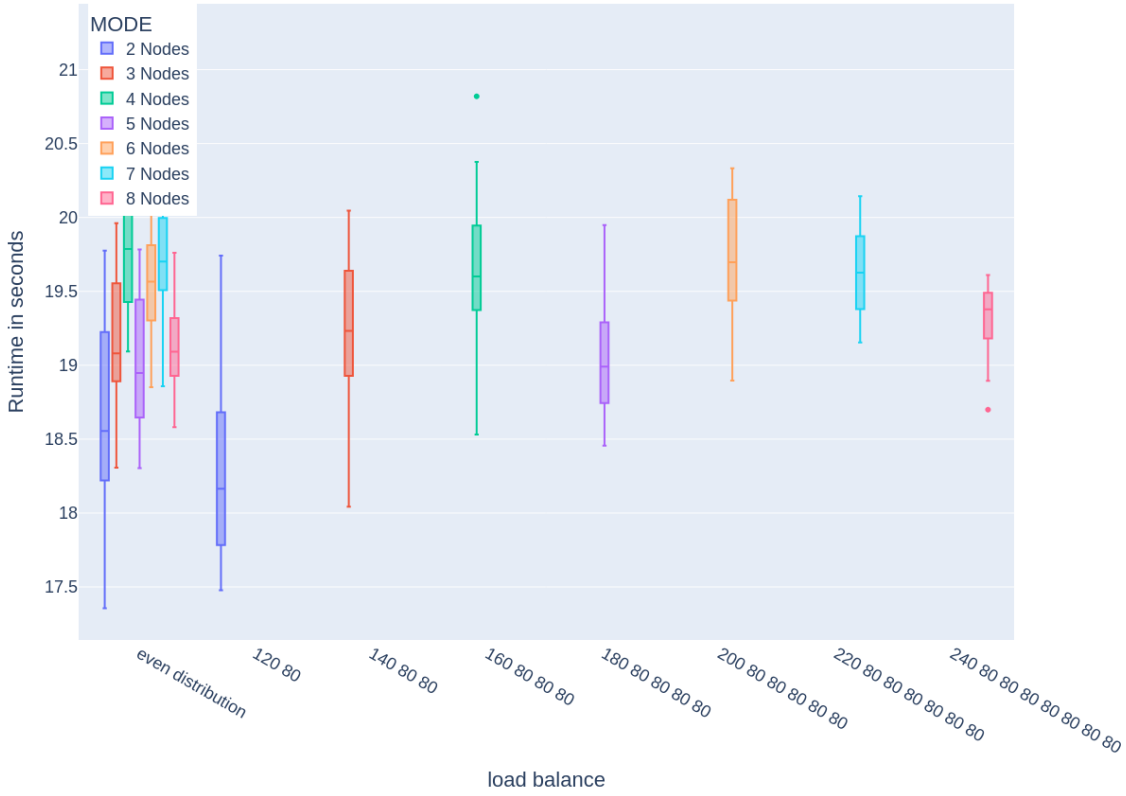


Figure A.4.: Results of the Benchmark 4.2.4. Lower is better.

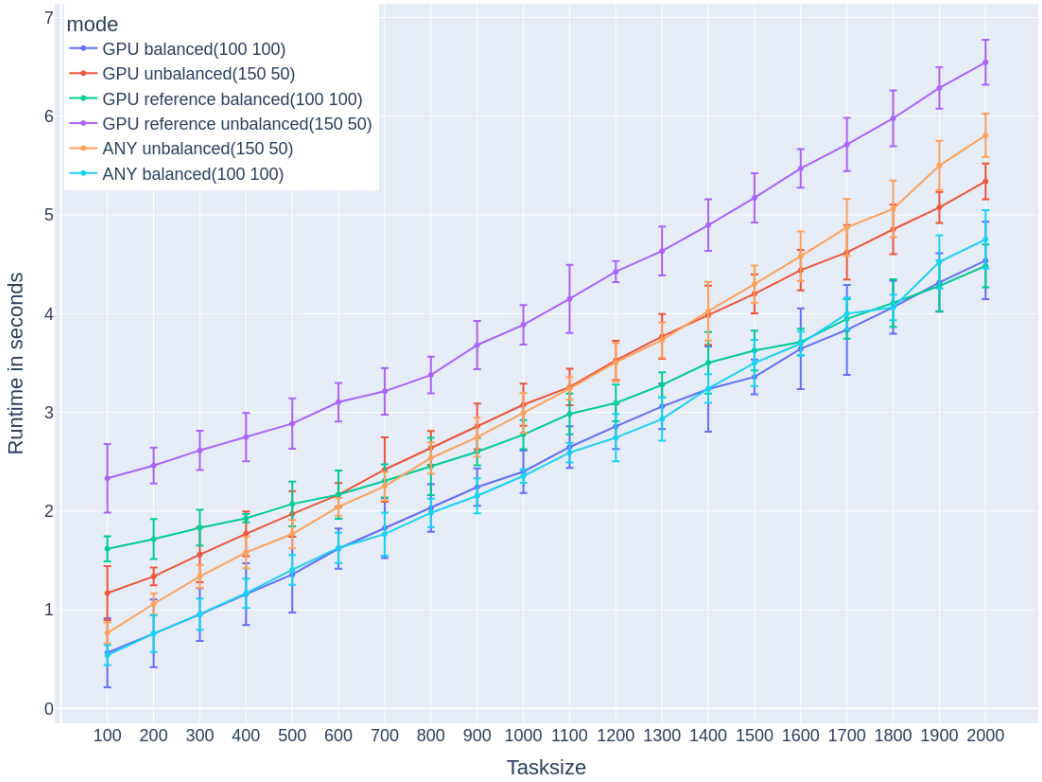


Figure A.5.: Results of the Benchmark 4.2.5. Lower is better.

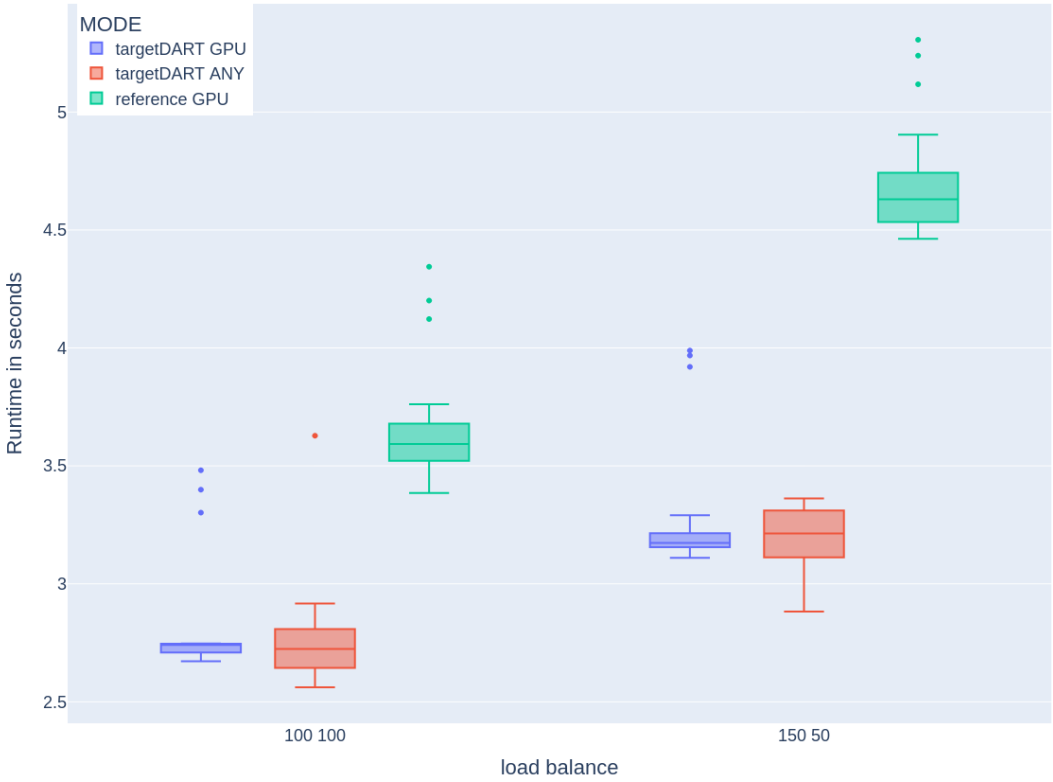


Figure A.6.: Results of the Benchmark 4.2.6. Lower is better.

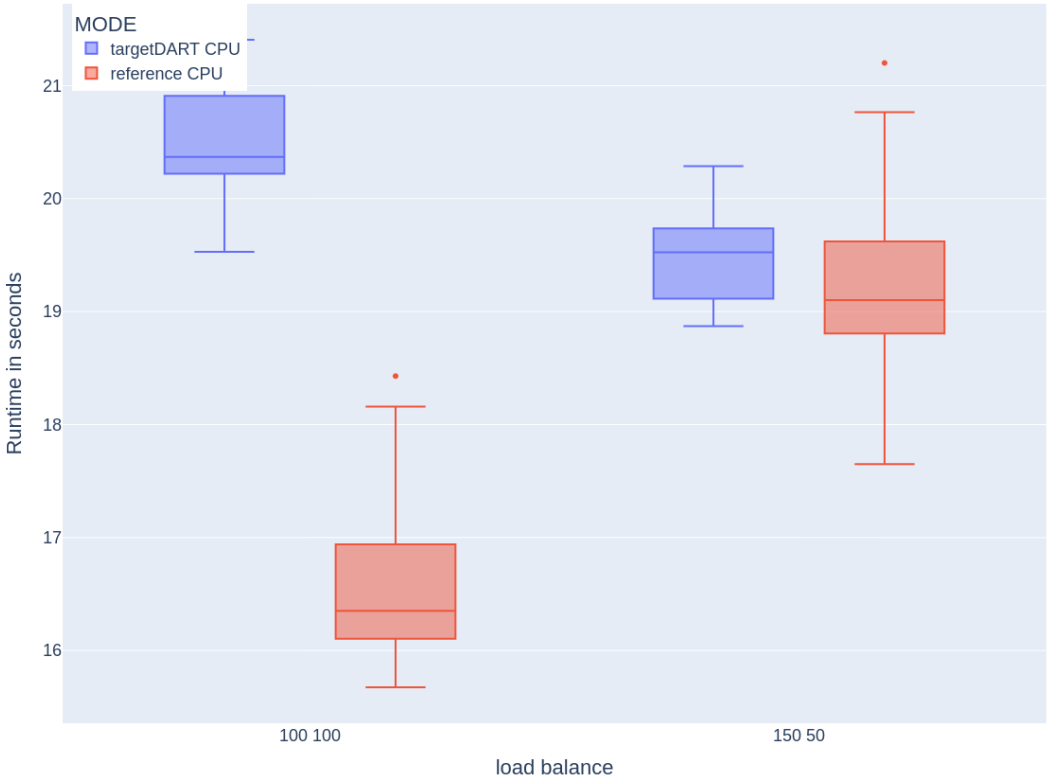


Figure A.7.: Results of the Benchmark 4.2.6. Lower is better.

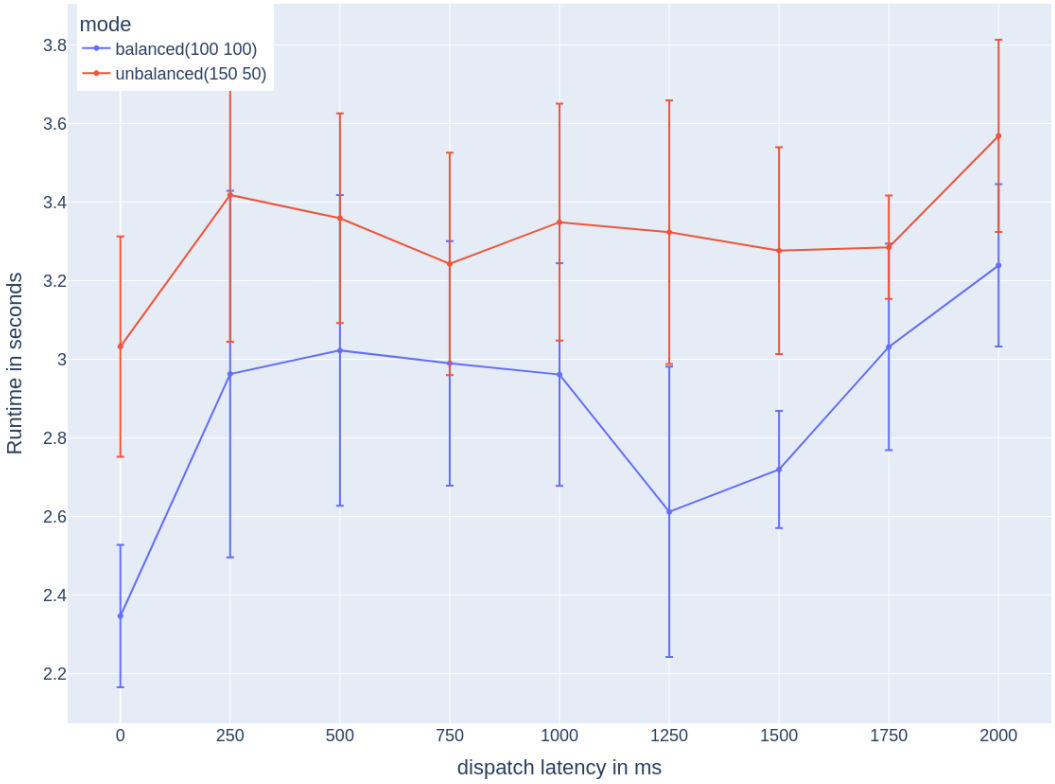


Figure A.8.: Results of the Benchmark 4.2.7. Lower is better.

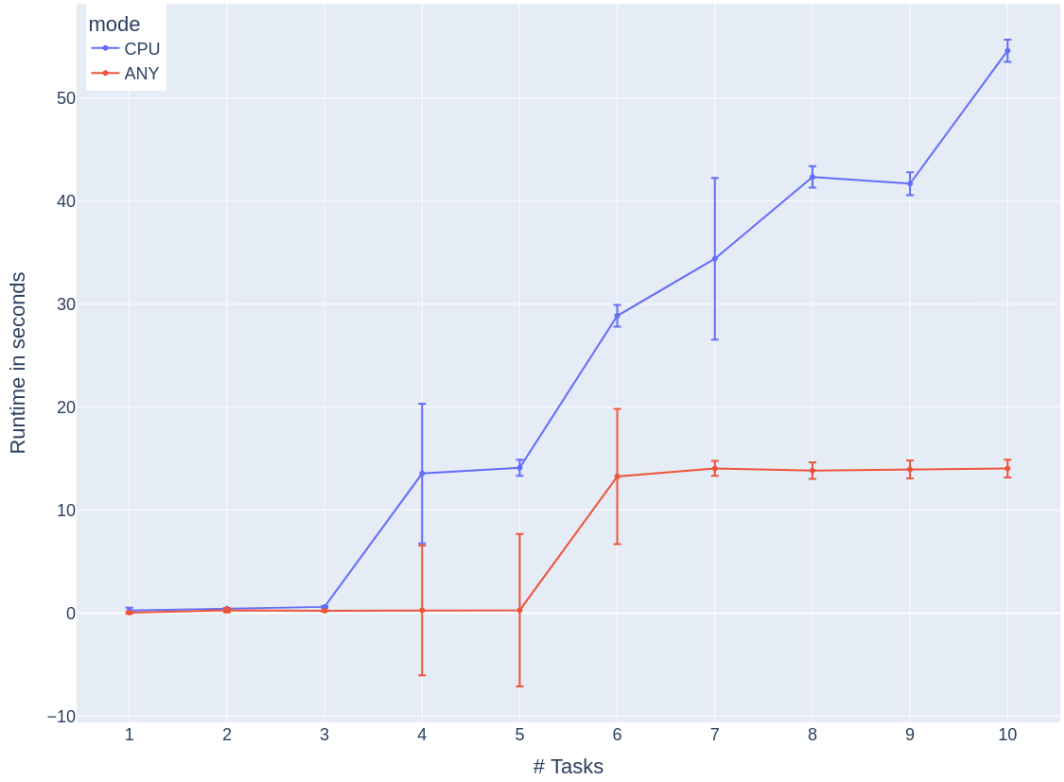


Figure A.9.: Results of scaling the number of tasks according to 4.2.8, but with the bug that increases the runtime if a node starts with 0 tasks. Lower is better.

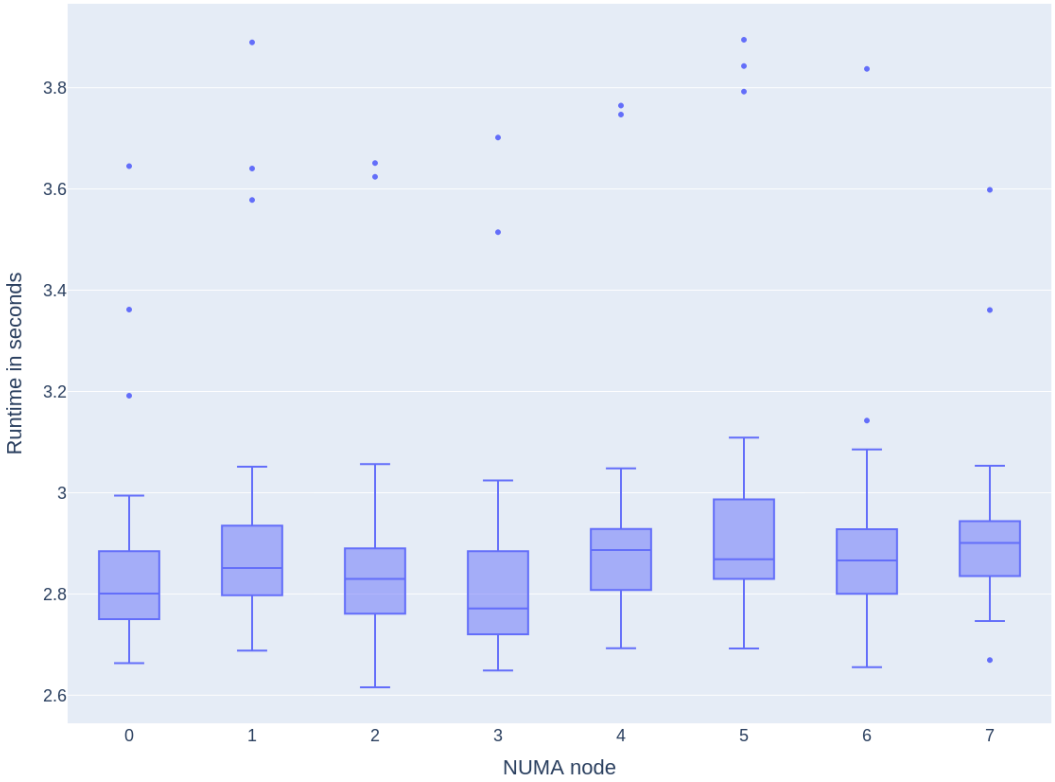


Figure A.10.: Results of the Benchmark 4.3.3. Lower is better.

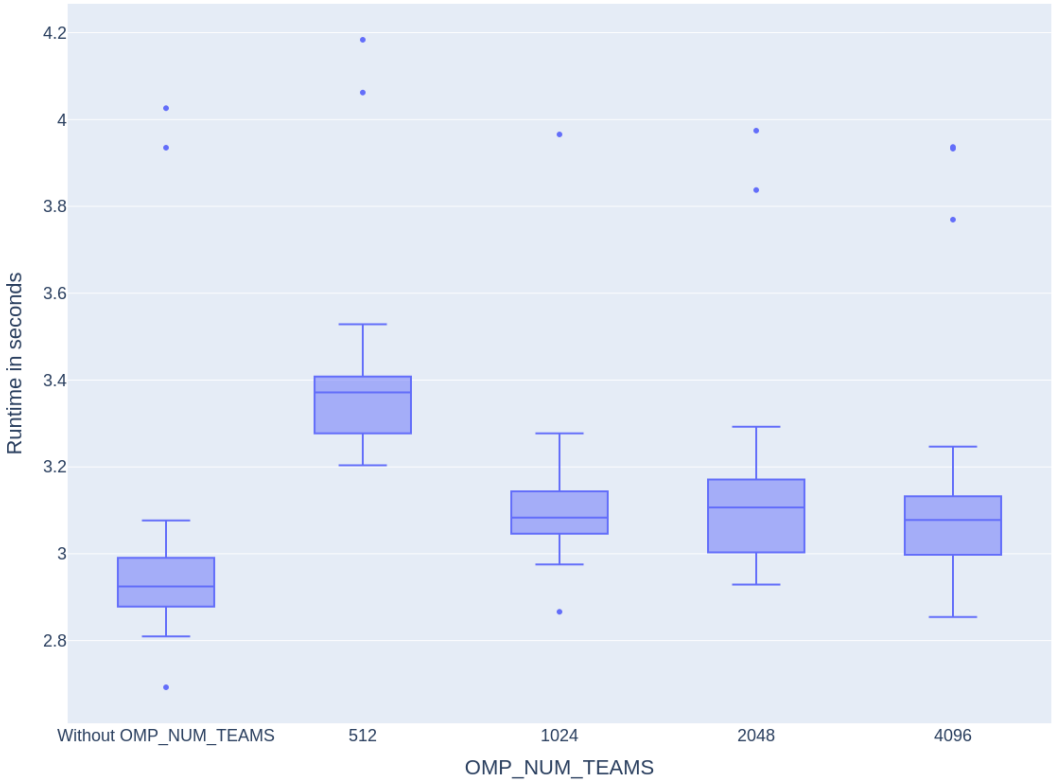


Figure A.11.: Results of the Benchmark 4.3.5. Lower is better.

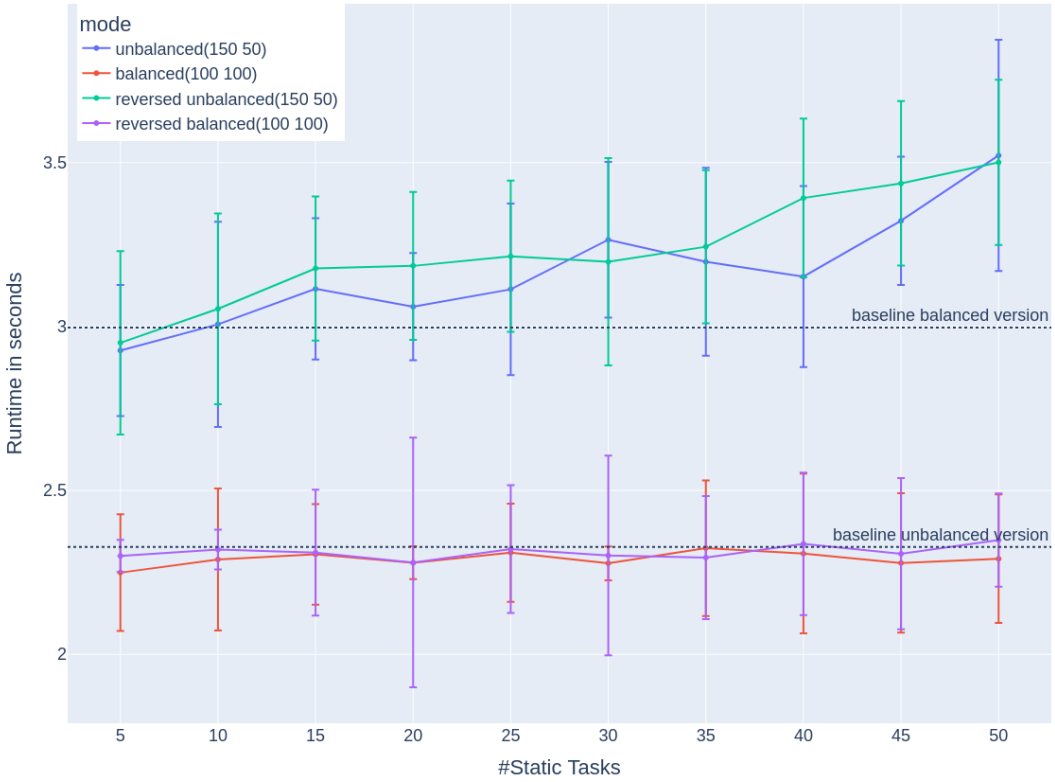


Figure A.12.: Results of the Benchmark 4.3.6. Lower is better.

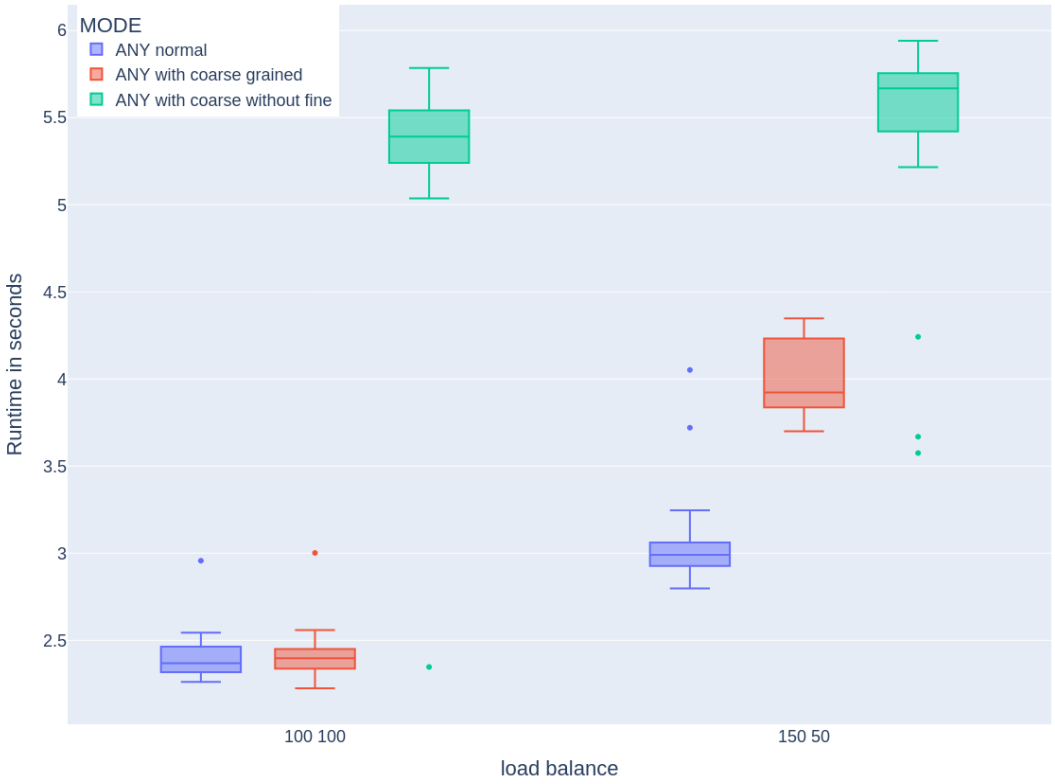


Figure A.13.: Testing different configurations for the scheduling algorithm. Results for the targetDART ANY device. Lower is better.

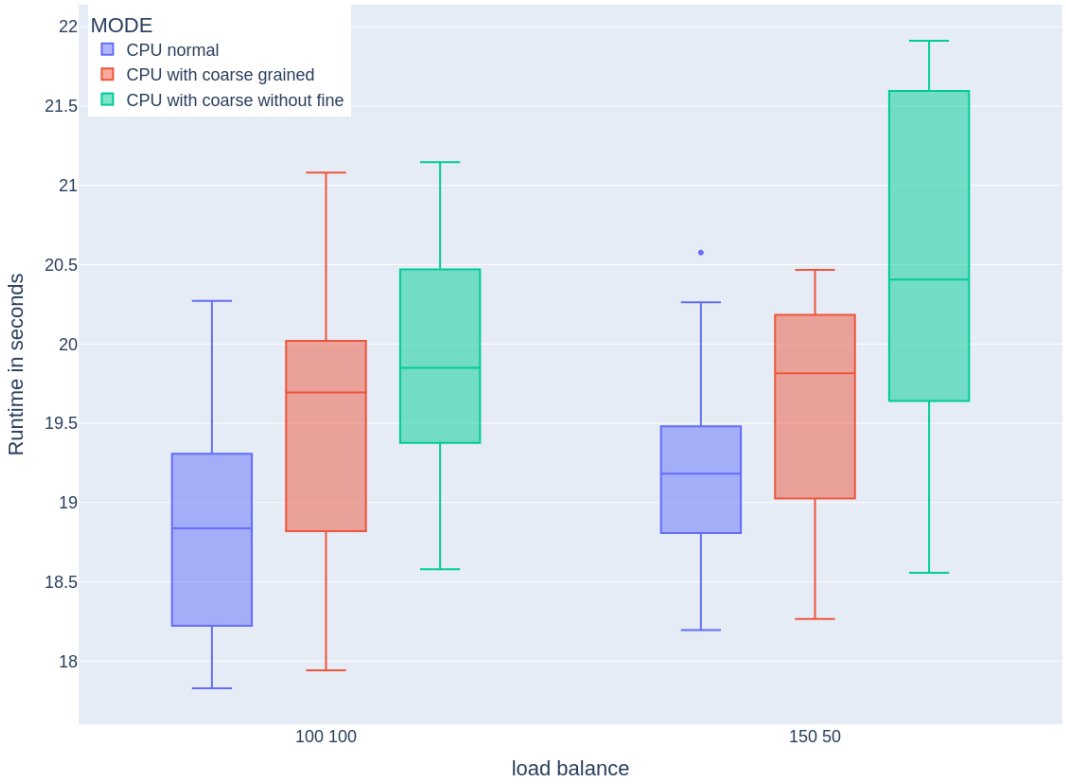


Figure A.14.: Testing different configurations for the scheduling algorithm. Results for the targetDART CPU device. Lower is better.

Bibliography

[1] H. Kaiser, T. Heller, A. Berge, and B. Adelstein-Lelbach, "HPX V0.9.11: A general purpose C++ runtime system for parallel and distributed applications of any scale," 2015, http://github.com/STEllAR-GROUP/hpx. [Online]. Available: http://dx.doi.org/10.5281/zenodo.33656.

[2] Definition of the memory wall. Accessed 15.08.2025. https://link.springer.com/rwe/10.1007/978-0-387-09766-4_234.

[3] Source code of the targetDART plugin. Accessed 30. July 2025. https://github.com/targetDART/llvm-project.

[4] The OpenMP documentation about the hyperparameter OMP_NUM_TEAMS. Accessed 27. July 2025. https://www.openmp.org/spec-html/5.1/openmpse80.html.

[5] Specifications of Claix23 nodes. Accessed 15. July 2025. https://help.itc.rwth-aachen.de/en/service/rhr4fjjutttf/article/fbd107191cf14c4b8307f44f545cf68a/.

[6] Documentation of the random python library. Accessed 01.09.2025. https://docs.python.org/3/library/random.html.

[7] Marsha J Berger and Joseph Oliger. "Adaptive mesh refinement for hyperbolic partial differential equations". In: *Journal of Computational Physics* 53.3 (1984), pp. 484–512. ISSN: 0021-9991. DOI: https://doi.org/10.1016/0021-9991(84)90073-1. URL: https://www.sciencedirect.com/science/article/pii/0021999184900731.

[8] Anthony Cabrera et al. "Toward Performance Portable Programming for Heterogeneous Systems on a Chip: A Case Study with Qualcomm Snapdragon SoC". In: *2021 IEEE High Performance Extreme Computing Conference (HPEC)*. 2021, pp. 1–7. DOI: 10.1109/HPEC49654.2021.9622794.

[9] Norihisa Fujita et al. "CHARM-SYCL & IRIS: A Tool Chain for Performance Portability on Extremely Heterogeneous Systems". In: *2024 IEEE 20th International Conference on e-Science (e-Science)*. 2024, pp. 1–10. DOI: 10.1109/e-Science62913.2024.10678717.

[10] Daniel F. Harlacher et al. "Dynamic Load Balancing for Unstructured Meshes on Space-Filling Curves". In: *2012 IEEE 26th International Parallel and Distributed Processing Symposium Workshops & PhD Forum*. 2012, pp. 1661–1669. DOI: 10.1109/IPDPSW.2012.207.

[11] Yoshua Bengio James Bergstra. "Random Search for Hyper-Parameter Optimization". In: *Journal of Machine Learning Research* 13 (2012), pp. 281–305. URL: https://www.jmlr.org/papers/volume13/bergstra12a/bergstra12a.pdf.

[12] David A. Patterson John L. Hennessy. *Computer Architecture. A Quantitative Approach*. Morgan Kaufmann Publishers, 2019, pp. 372–373. ISBN: 9780128119051.

[13] Beau Johnston et al. "IRIS: Exploring Performance Scaling of the Intelligent Runtime System and its Dynamic Scheduling Policies". In: *2024 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. 2024, pp. 58–67. DOI: 10.1109/IPDPSW63119.2024.00017.

[14] Torben Kalkhof and Andreas Koch. "Speeding-Up LULESH on HPX: Useful Tricks and Lessons Learned using a Many-Task-Based Approach". In: *SC24-W: Workshops of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 2024, pp. 1223–1235. DOI: 10.1109/SCW63240.2024.00164.

[15] Zahra Khatami, Hartmut Kaiser, and J. Ramanujam. "Using HPX and OP2 for Improving Parallel Scaling Performance of Unstructured Grid Applications". In: *2016 45th International Conference on Parallel Processing Workshops (ICPPW)*. 2016, pp. 190–199. DOI: 10.1109/ICPPW.2016.39.

[16] Zahra Khatami et al. "A Massively Parallel Distributed N-body Application Implemented with HPX". In: *2016 7th Workshop on Latest Advances in Scalable Algorithms for Large-Scale Systems (ScalA)*. 2016, pp. 57–64. DOI: 10.1109/ScalA.2016.012.

[17] Jungwon Kim et al. "IRIS: A Portable Runtime System Exploiting Multiple Heterogeneous Programming Systems". In: *2021 IEEE High Performance Extreme Computing Conference (HPEC)*. 2021, pp. 1–8. DOI: 10.1109/HPEC49654.2021.9622873.

[18] Jannis Klinkenberg et al. "CHAMELEON: Reactive Load Balancing for Hybrid MPI+OpenMP Task-Parallel Applications". In: *Journal of Parallel and Distributed Computing* 138 (2020), pp. 55–64. ISSN: 0743-7315. DOI: https://doi.org/10.1016/j.jpdc.2019.12.005. URL: https://www.sciencedirect.com/science/article/pii/S0743731519305180.

[19] Het Mankad et al. "A Performance-Portable MultiGPU Implementation of 3D Euler Equations using ProtoX and IRIS". In: *SC24-W: Workshops of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 2024, pp. 1723–1731. DOI: 10.1109/SCW63240.2024.00215.

[20] Het Mankad et al. "ProtoX: A First Look". In: *2023 IEEE High Performance Extreme Computing Conference (HPEC)*. 2023, pp. 1–6. DOI: 10.1109/HPEC58863.2023.10363547.

[21] G.R. Mudalige et al. “OP2: An active library framework for solving unstructured mesh-based applications on multi-core and many-core architectures”. In: *2012 Innovative Parallel Computing (InPar)*. 2012, pp. 1–12. DOI: 10.1109/InPar.2012.6339594.

[22] Narasinga Rao Miniskar et al. “IRIS-BLAS: Towards a Performance Portable and Heterogeneous BLAS Library”. In: *2022 IEEE 29th International Conference on High Performance Computing, Data, and Analytics (HiPC)*. 2022, pp. 256–261. DOI: 10.1109/HiPC56025.2022.00042.

[23] Anne Reinarz et al. “ExaHyPE: An engine for parallel dynamically adaptive simulations of wave problems”. In: *Computer Physics Communications* 254 (2020), p. 107251. ISSN: 0010-4655. DOI: https://doi.org/10.1016/j.cpc.2020.107251. URL: https://www.sciencedirect.com/science/article/pii/S001046552030076X.

[24] Philipp Samfass et al. “Lightweight task offloading exploiting MPI wait times for parallel adaptive mesh refinement”. In: *Concurrency and Computation: Practice and Experience* 32.24 (2020), e5916. DOI: https://doi.org/10.1002/cpe.5916. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/cpe.5916. URL: https://onlinelibrary.wiley.com/doi/abs/10.1002/cpe.5916.

[25] Adrian Schmitz et al. “targetDART: Dynamic Migration of OpenMP GPU Kernels in Heterogeneous Clusters”. In: *Submitted to: 31st ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming*. ACM. 2018.

[26] Christian Siebert and Felix Wolf. “Parallel Sorting with Minimal Data”. In: *Recent Advances in the Message Passing Interface*. Ed. by Yiannis Cotronis et al. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 170–177. ISBN: 978-3-642-24449-0.

[27] Ericson Marquiere Reis Silva and Henrique Cota de Freitas. “Task Scheduling for Autonomous Vehicles with Heterogeneous Processing via Integration of the CARLA Simulator and StarPU Runtime”. In: *2025 IEEE International Conference on Consumer Electronics (ICCE)*. 2025, pp. 1–6. DOI: 10.1109/ICCE63647.2025.10929988.

[28] Mathialakan Thavappiragasam et al. “Addressing Load Imbalance in Bioinformatics and Biomedical Applications: Efficient Scheduling across Multiple GPUs”. In: *2021 IEEE International Conference on Bioinformatics and Biomedicine (BIBM)*. 2021, pp. 1992–1999. DOI: 10.1109/BIBM52615.2021.9669317.

[29] G. Vossen W. Oberschelp. *Rechneraufbau und Rechnerstrukturen. 10. Auflage*. Oldenburg Wissenschaftsverlag GmbH, 2006, pp. 328–329. ISBN: 3486578499.

[30] Xiaoyang Wang et al. “Dynamic GPU Scheduling With Multi-Resource Awareness and Live Migration Support”. In: *IEEE Transactions on Cloud Computing* 11.3 (2023), pp. 3153–3167. DOI: 10.1109/TCC.2023.3264242.