DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatics

Exploring Load Balancing for Modern Numerical Software

Maximilian Karpfinger

DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatics

Exploring Load Balancing for Modern Numerical Software

Evaluation von Lastbalancierungsverfahren in Moderner Numerischer Software

Author:	Maximilian Karpfinger
Supervisor:	Univ.-Prof. Dr. Michael Bader
Advisor:	Philipp Samfaß
Submission Date:	16.03.2020

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

Munich, 16.03.2020 Maximilian Karpfinger

Acknowledgments

I would like to express my sincere thanks to my advisor Philipp Samfaß. Without his constant support and guidance this thesis would not have been possible. I also appreciate the regular meetings where I have learned a lot.
I also want to thank my family and friends for supporting me.

Abstract

Solving hyperbolic differential equations is a very important field in high performance computing, yet it is not only highly time demanding, but also critical for resources as memory and energy. The more important is an efficient numerical solution to such problems, since an increase in computation time also leads to an increase in energy consumption. However, these numerical schemes might force load imbalances which cause wait-times. Causes for such inequalities have to be detected and fixed. In this work the root for imbalances will be explored and their impact on the total run-time will be investigated. The motivation is to reduce these by implementing load balancing schemes that try to balance load distribution at run-time. Such an approach which is based on MPI-wait-times will be examined in this thesis for different scenarios in the ExaHyPE engine where an improvement in the overall elapsed wall-time of one third was measured.

List of Abbreviations

abbreviation	meaning
ADER	arbitrary high-order derivative
CCZ4	CCZ4 single black hole
DG	discontinous Galerkin
FTS	Fused-Time-Step
FV	Finite Volume
HPC	High Performance Computing
MPI	Message Passing Interface
PDE	partial differential equation
STP	Space-Time-Prediction
TBB	Threading-Building-Blocks

Table 1: List of all abbreviations used in this thesis

Contents

1 Introduction

High-Performance-Computing is one of the most important fields in modern science. Common home computers cannot solve problems with a high complexity order fast, instead HPC provides an answer to complex tasks as the solution of partial differential equations or graph problems. They are not only important in science, but also for economical and governmental aspects. For example the spread of disease or predicting natural disasters as tsunamis or hurricanes[1].
Since the technological development of CPUs seems to reach the bounds of Moore's law which states that the number of transistors on a central processing unit doubles frequently (12-24 months) no longer holds[2], researchers need a different approach to have the possibility of scaling-up the available computing power. Such an advancement is the shift from one CPU to multi-core CPU architectures. These provide several computing cores which can be utilized to achieve parallelism thereby enabling scalability on a high degree. However, programming parallel application is a very complex assignment. Even with a sophisticated approach, one might still run into problem due to load imbalances which cause a CPU to have a sub-optimal utilization. The first chapter gives a background to understand the basics of parallel programming. Going on with the second chapter, the ExaHyPE engine which solves differential equations is going to be introduced. In the third chapter a profound look into load imbalances and approaches to handle them is taken. The fourth chapter is about the implementation of the instrumentation for the experiments. Finally, in the fifth chapter, several experimental result will be presented that show where load distribution inequalities take place and how efficiently they can be tackled by trying to load balance at run-time.

2 Background

This chapter shall give a short introduction into parallel programming and present two common building blocks.

2.1 Message Passing Interface

A common way to achieve parallelism in modern applications is to deploy multiple compute nodes. All processes have separate memory address space so in order to establish data exchange via communication between the processes the message passing interface *MPI* can be used. It is a library which can be implemented in the C language. For communication every process needs a unique identifier number, henceforth will be referred to as a *rank*. The two most common MPI library calls are MPI_Send and MPI_Recv which, as their name indicates are needed to send and receive data from other ranks. For example to send data from rank zero to rank one, first of all rank one collects all data in a buffer. Although, before the data gets sent, rank one has to acknowledge that it wants to receive data from rank zero. After the data is received, a confirmation to rank zero is sent back[3]. One goal of MPI is to accomplish efficient communication by avoiding memory to memory copies[4]. One is able to limit the thread environment as shown in the following table.

level	explanation
MPI_THREAD_SINGLE	Only one thread will execute
MPI_THREAD_FUNNELED	The process may be multi-threaded, application must ensure that only the main thread makes MPI calls
MPI_THREAD_SERIALIZED	The process may be multi-threaded, multiple threads may make MPI calls, but only one at a time
MPI_THREAD_MULTIPLE	Multiple threads may call MPI, with no restrictions

Table 2.1: Information retrieved from https://www.mpi-forum.org/docs/mpi-3.1/mpi31-report/node303.htm

MPI introduced these to ensure thread-safety levels. This way race-conditions can be prohibited which is important because when two ranks access and write the same data from a non thread-safe data-structure, then the application behaviour might be undesired. For example, when two ranks concurrently write different values to a variable, then it might happen that the first write will be irrelevant since the second write simply overwrites the first value. A very important metric is the MPI wait-time which measures the elapsed wait time of a CPU in MPI library calls. Usually, load imbalances can be seen from this because MPI is waiting for communication operations of the ranks.

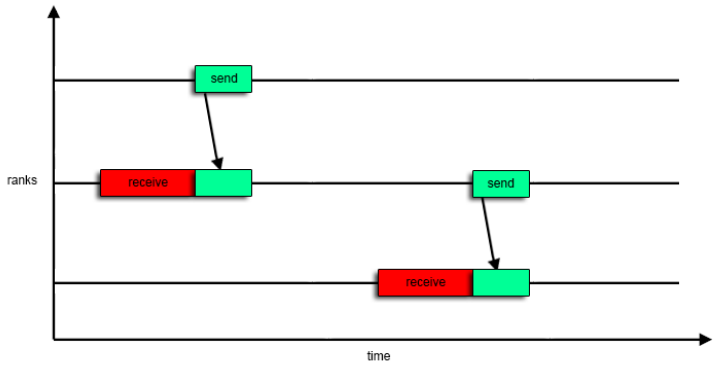


Figure 2.1: Late-sender pattern illustrating MPI-wait times.

Figure 2.1 shows how MPI wait times are caused due to ranks waiting in MPI_Recv (red) for data to be sent with MPI_Send (green) from other ranks. In chapter 4 an approach that tackles these wait-times is going to be introduced.

2.2 Intel Threading Building Blocks

Threading Building Blocks is an open source library that helps programmers to circumvent the complexity of writing correct parallel code with low level threads[5]. As with processes, each thread is identified by a unique number and belongs to exactly one rank. Since the main goal in high performance computing applications is to achieve scalability, *TBB* is one of the more widely-used libraries among *OpenMP* in modern numerical software. The main characteristic of the programming model of TBB is that it uses higher level tasks which then get mapped to threads[5]. To achieve this, programmers shall decompose a problem into tasks and are also able to use provided parallel for loops or concurrent data-structures such as queues or hashmaps.

3 ExaHyPE Engine

ExaHyPE as an exascale hyperbolic PDE Engine was created to provide solutions to first-order hyperbolic partial differential equations[6]. The engine controls the process of the simulation, similar to a game engine which is, for example, responsible for the physics of the game. PDEs are needed for the simulation of seismic events as for example earthquakes or tsunamis. In the field of astrophysics ExaHyPE helps with the search of gravitational waves emitted by binary neutron stars[6]. Another aspect of the engine is that one can generate output of the simulation results and visualize it with software as ParaView[7] and thus gain knowledge about real physical scenarios like black holes. From [6], the equation of a generalized first-order form of the PDEs solved in ExaHyPE may be written as:

$$P\frac{\partial}{\partial t}Q + \nabla \cdot F(Q) + B(Q) \cdot \nabla Q = S(Q) + \sum_{i=1}^{n_p s} \delta_i \tag{3.1}$$

by ignoring non-conservative parts and sources it can be simplified to:

$$P\frac{\partial Q}{\partial t} + \nabla \cdot F(Q) = 0 \tag{3.2}$$

whereas Q represents the state vector of the variables of the computational domain and F is the flux of energy in space-time.

3.1 Peano

Peano is a framework that is used by ExaHyPE and responsible for the management of the grid which represents the computational domain of a simulation. Peano makes use of a *master-worker* architecture where rank zero takes the role of the supervising master. Hence rank zero will distribute the work load to all other ranks which will do computing jobs and communicate with each other. An illustration for that can be found in 3.1. In Peano data is stored in cells which are accumulated in *space-trees*[8]. These are used to distribute grid work to the worker ranks whereas the Cartesian grid's dimension is either two or three. Another duty of Peano is to set up the shared and distributed memory environment. For distributed memory MPI and in the following TBB for shared memory is deployed.

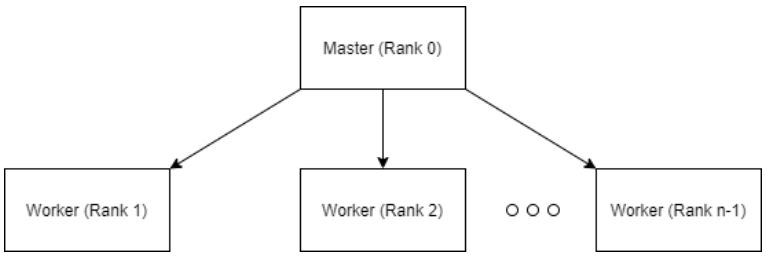


Figure 3.1: Master-worker pattern where rank zero distributes tasks to other ranks

3.2 Computational Domain

The computational domain stores the settings of a simulation scenario, whereas it consists of several attributes.

attribute	description
width	sets the size of the domain
offset	sets the offset of the domain
outside cells	sets the number of outside cells
dimension	sets the dimension of the domain
time steps	sets the number of time steps of the simulation

Table 3.1: Important parameters of the computational domain considered for the experiments.

Another important aspect is the order and the maximum mesh size of the used solver. The mesh size determines the grid size. Notably, only grids with a mesh of 27x27x27 or 81x81x81 are going to be examined in the experiments in chapter 6 as they are a balanced problem decomposition for Peano. For experiments with 28 nodes it will be mentioned how many cores each rank gets. Finally, outside cells shall be explained. Outside cells represent the cells that are the border of the whole grid. Setting outside cells to 1 will shorten the actual grid by one cell on each border, whereas one might distinct outside cells for the left and the right side of the grid.

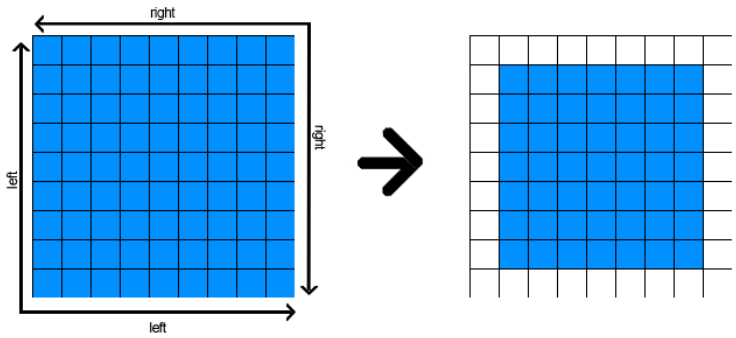


Figure 3.2: Changing a 2D computational domain from zero to one outside cells on both sides.

The for computation relevant parts of a 9x9 grid are represented by blue cells, while white cells are omitted from computation. Information about the computational domain is stored together with the characteristics of the deployed solver and the settings for shared/distributed memory in a single specification file which is used by the ExaHyPE toolkit to generate code. This generated code is then compiled to an executable that takes the specification file as an argument to solve the given problem.

3.3 ADER-DG Solver

The ADER-DG solver grants a numerical solution to PDEs by deploying a time-stepping scheme. Each time-step is further divided into three smaller steps which shall be explained in the following. All formulas are taken from[9]. First, again with ignoring non-conservative terms and sources to get a simpler formulation of a time-step. Non-conservative terms could represent the effect of for example heat or friction and sources describe external forces.

$$\int_c v_h D_h dx = \int_T^{T+\Delta T} \int_c \nabla v_h : F(Q_h^*) dx dt - \int_T^{T+\Delta T} \oint_{\partial_c} v_h [F(Q_h^*) \cdot n]_{\partial_c} ds dt \tag{3.3}$$

ADER replaces the original variables state vector Q of 3.1 by a predicted solution Q^* of the space-time and integrates over the cells c to get an update for the time-step ΔT. In the first step, which will be called STP or space-time prediction, the solutions for cells are calculated independently which means that the effects of neighboring cells are omitted:

$$\forall c \in \mathrm{T} : \int_T^{T+\Delta T} \int_c \left(\frac{\partial Q_h^*}{\partial t} + \nabla \cdot F(Q_h^*) \right) \varphi_h \, dx \, dt = 0 \tag{3.4}$$

Since nonlinear PDE systems are solved here, Picard iterations are used for the solution. They will take an important role in the following chapter 4 and in chapter 6 as they

are computationally expensive and generally make up the most time of the overall wall-time of a simulation [9]. Thus they are the main focus for instrumentation in this thesis. In addition to the prediction, the flux is also calculated because it is needed for the next step. In the second step, also known as the Riemann problem, jumps of the flux from the first step which are caused by projecting the predicted solution Q^* onto the cells are counteracted for example by applying a Rusanov solver as in the work[9]:

$$\forall c \in \mathrm{T} : \left[F(Q_h^*) \cdot n\right]_{\partial_c} = F_h^* \; with \; F_h^* = F_h^*(Q_h^{*\pm}, F(Q_h^*)^{\pm} n) \tag{3.5}$$

Both sides ($\pm$) of the predicted solution Q^* and the flux $F(Q^*)$ are taken as an input to gain flux contributions of the time-step for the cells' faces. n represents the normal vector that is used for the scalar product with the flux $F(Q^*)$.

In the third step, also known as the correction or update step, the results are formed by combining the first two steps. Specifically, the Riemann solution is added to the predicted values as a correction as through this Q is gained. Another setting in the file from the optimization's section, which is important for the experiments, is spawn_update_as_background_thread. Through that one can enable the update step to be done in the background.

Notably, for this thesis a nonlinear ADER-DG will be used which is important due to the existence of Picard iterations in the experiments if not explicitly mentioned otherwise. Among ADER-DG, one might choose a FV solver which can also be combined with a limiter. Basically, the idea of a limiter is to detect troubled cells where further computation is needed which happens when the ADER-DG solver fails to compute correct results due to the presence of shock waves[10]. The recalculation is done with a FV Godunov solver which tends do be computationally heavy to accomplish high accuracy as described in the paper[6].

3.4 Simulation Scenarios

Among many simulation scenarios in ExaHyPE, only the three following basic scenarios will be consulted for the experiments in the last chapter 6. All of them deploy the previously introduced ADER-DG solver. First, the Euler scenario describes solid and fluids dynamics which are important in fields as metallurgy[11]. Second, Gauge is a trivial artificial benchmark and grants no further useful results as it was created to test the correctness of the implemented CCZ4[12] which is based on a hyperbolic first-order form of the Einstein equations[13]. Third, the single black hole which relies on CCZ4 will be referred to as CCZ4 in the following.

On the following page the variable g_{yy} of the 3-metric of Q for the single black hole scenario with a limiter at time-step t=0.01 and t=0.02 is plotted. In combination with

the other components of the 3-metric one can calculate the intrinsic curvature of space which is caused by the black hole.

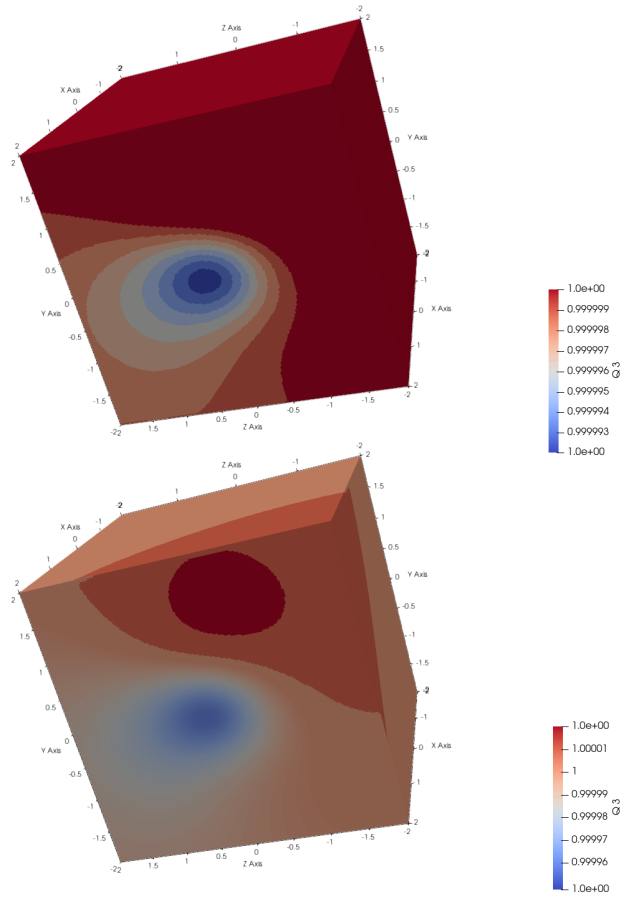


Figure 3.3: Single black hole 3-metric tensor g_{yy}

4 Approaches for Load Balancing

In this chapter the definition for load imbalances is explained. Furthermore, causes for workload inequalities are discussed. Finally, two frameworks for load balancing are examined.

4.1 Load Imbalance Metric

Given a set R of ranks with $|R| > 1$ load imbalances might appear. For given ranks i, $j \in R$ an imbalance exists whenever rank i has to wait for data from rank j and vice versa, since in a perfect load distribution rank i and j would have an equivalent amount of work. Such imbalances may be calculated as the empirical variance or the standard deviation of the time spent in an instrumented region per rank. However, a better metric is:

$$\frac{(|R| - 1) \times max(r_t \in R)}{\sum_{i=1}^{|R|-1} r_{i_t}} \tag{4.1}$$

Where r_t is the overall time spent in STPs of rank r. Because of the Master-Worker architecture in ExaHyPE, rank zero as the master process will be omitted from instrumentation. Thus, the cardinal number of the set of ranks is reduced by one. The rank with the highest altogether STP time is selected and its value is divided by the average time spent in STPs of all ranks. This grants a metric that describes a scaled difference between the maximum and the average STP time of the ranks.

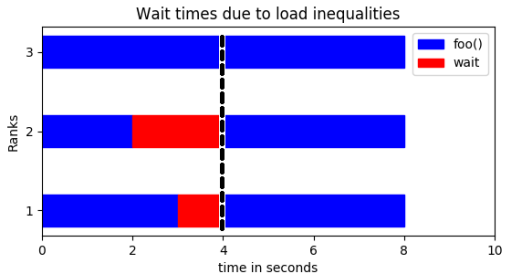


Figure 4.1: Illustration of wait-times caused by load imbalances

For the example of Figure 4.1 the introduced metric would be calculated as follows.

$$\frac{(|R|-1) \times max(r_t \in R)}{\sum_{i=1}^{|R|-1} r_{i_t}} \tag{4.2}$$

$$= \frac{3 \times 4s}{9s} = 1.\overline{3} \tag{4.3}$$

4.3 yields that the rank with the highest time spent in the first call to foo() takes one third time longer than the average rank. This causes a combined wait-time of three seconds. A better work distribution for this example would be to offload the workload equal to one second from rank three to rank two. In a hypothetical system where communication and offloading costs are negligible, the total wait time would be reduced to zero and the overall elapsed run-time decreased to seven instead of eight seconds.
There are different causes for load imbalances. Even for a hypothetical perfectly balanced problem, disproportions can still occur due to instabilities of hardware. For example, a CPU might has to reduce its clock speed because of heat, rendering its computing power lower[2]. These causes for imbalances are not further examined in this thesis and unevenness in software will be examined instead. For this, ExaHyPE will be utilized. On one hand, imbalances in ExaHyPE can be caused by deploying outside cells which will make the distribution of grid works to ranks naturally unequal. This will be referred to as an imbalanced domain. On the other hand, numerical schemes might cause waiting times, which will be referred to as a numerical imbalance which can be found in Chapter 2 discussed STPs, specifically due to Picard iterations. From the ADER-DG kernels spaceTimePredictorNonlinear class for three dimensions:

Algorithm 1 aderPicardLoopNonlinear

```
1:  function aderPicardLoopNonlinear
2:      if useSolversMaxPicardIterations then
3:          maxPicardIterations ← solverMaxPicarditerations
4:      else
5:          maxPicardIterations ← (order + 1) × 2
6:      for i < maxPicardIterations do
7:          //do compute heavy stuff
8:          if solution is reasonable then
9:              break
10:     return i
```

As illustrated in Algorithm 1, it is clear why the run time of the function can be different

since the solution might be reasonable in a different amount of iterations, thus causing numerical imbalances.
Concluding, load imbalances cause several serious problems in high performance computing as they reduce the positive run time effects of up-scaling hardware due to causing wait times. Another important aspect is that with a higher elapsed time more energy is needed, so with a proper workload distribution a decrease of energy consumption can be achieved. In the following two approaches for run-time workload balancing are introduced.

4.2 AMPI/Charm++

First, Charm++ is a object oriented parallel programming system which deploys migratable objects thereby enabling its core feature to dynamically load balance at run-time. Another characteristic is its asynchronous communication pattern. The key idea behind the programming model is into split the whole program to smaller work and data units (Chares) which can be assigned to any given processor[14]. AMPI is an MPI implementation on top of Charm++[15]. The difference to MPI is that a user can provide more ranks than the actual amount of physical cores by virtualization. These ranks are implemented as user-level threads and can exchange their data through Charm++ enabling load balancing. The advantage of AMPI is that it can be built with any MPI application with only needing to change global and static variables to thread local variables[16]. Load balancing in AMPI is explained in the following figure.

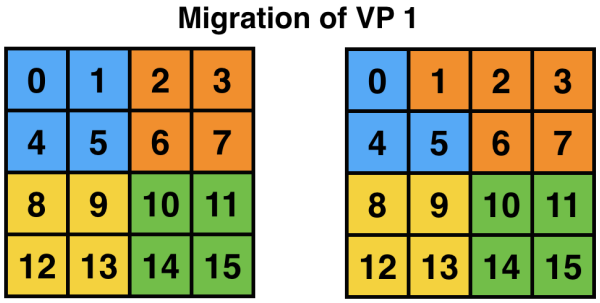


Figure 4.2: Image from https://charm.readthedocs.io/en/latest/_images/migrate.png shows the migration of a virtual process

In the first step the problem is over-composed by creating more virtual ranks than actual available processors. Every color represents one allocatable core which is over-composed to four ranks. For the example of migrating one of these virtual ranks the blue core migrates its rank one to the orange core which might be helpful when the

blue processing unit has a higher amount of workload and causes wait-times for others.

4.3 Task Offloading

The main idea of the in [17] introduced task-offloading approach is to make reactive load balancing decisions depending on MPI wait-times. It relies on the existence of tasks such as STP jobs, mentioned in chapter 3 2. Ranks with a high amount of tasks might detect that they are time demanding are referred to as critical ranks. These have the possibility to offload jobs to ranks that are idle (optimal victims) while not increasing the total run time. However, usually, when ranks become idle performance is already lost, so it would be preferred to proactively offload to ranks with a low amount of workload. Compared to the AMPI/Charm++ approach threads or tasks are not migrated but rather sent to other ranks with MPI. On these computation continues and results are returned to the original rank. As the definition of a victim rank states that no other rank waits for this specific rank, it might occur that a victim gets too many tasks from critical ranks which will cause an emergency. These are handled by blacklisting the overloaded victim, whereas every rank stores such a list. Blacklisted ranks will no longer receive tasks from critical ranks. Because of the nature of MPI wait-times it is possible to generate a wait-time-graph from which the critical rank can be deduced by a greedy graph traversal algorithm. The target is to iteratively remove the critical node from the graph such that, hypothetically, no more wait-times exist thus having a graph where no nodes are connected.

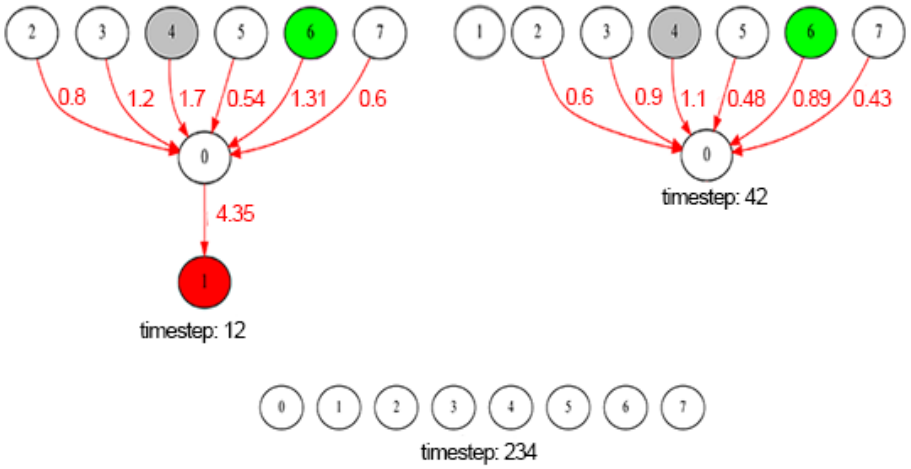


Figure 4.3: Change of waiting graph by task-offloading

Figure 4.3 shows a hypothetical run with task-offloading where at first rank one (red) is denoted as the critical rank as all other ranks have to wait for it. Rank six (green) is the optimal victim and will receive tasks to execute from critical ranks. Rank four (black) is marked as a blacklisted rank because it caused an emergency in a previous step. In the next time-step the red wait-times have decreased and rank 1 no longer delays the other ranks. Finally, in time-step 234 a perfect load distribution is achieved since no wait-time relations exist.
The approach is implemented in ExaHyPE and can be switched on/off via the specification file by setting the *offloading* option to *aggressiveHybrid* or *none* in the settings for distributed memory. In addition to the usual STP jobs, migratable STP jobs can now be spawned which can be sent to other ranks via MPI and will be handled there. As soon as they are finished the result will be returned back to the original rank. However, a distinction has to be made for skeleton cells which cannot be offloaded, since they are communication critical unlike enclave cells that can be handled in the background, i.e. are possible to be offloaded[18]. Finally, it is possible to change the setting of *offloading_progress* to *none* or *progress_task*. With progress_task enabled the request manager which stores MPI requests is polled more aggressively and critical tasks might be rescheduled with priority to reduce wait-times[17].

5 Implementation

This chapter discusses the applied instrumentation methodology which were used to collect data from the experiments.

5.1 Score-P 6.0

The Score-P performance measurement infrastructure for parallel code was created to create a joint infrastructure for several high performance computing tools. Through that the problems of using separate tools, such as redundancies, shall be reduced[19]. With Score-P it is possible to simply instrument functions using USER REGIONS. For each region Score-P collects information about wall-time and number of visits for each MPI Rank. Since Intel TBB is used, it is required to instrument with a Score-P version higher or equal to 5.0 because in lower versions instrumenting with TBB is not supported[20].

Algorithm 2 Score-P instrumentation

```
1: include "scorep/SCOREP_User.h"
2: function FOO
3:     SCOREP_USER_REGION_DEFINE( my_region_handle )
4:     SCOREP_USER_REGION_BEGIN( my_region_handle, REGION_NAME,
   SCOREP_USER_REGION_TYPE_COMMON )
5:     // do something
6:     SCOREP_USER_REGION_END( my_region_handle )
```

In Algorithm 2 the user region instrumentation via Score-P is explained. First, it is necessary to include the Scorep_User header for the Score-P wrapper to compile. To instrument a region, the user has to define a region handle and then begin measuring at the point of interest. The user also provides a region name and type. After termination, Score-P will create a cubex file which can either be viewed with the Cube-GUI or dump the information from it. The regions considered for instrumentation with Score-P are the function run() from ADERDG_PredictionJob where the elapsed time is measured and from ADER-DG kernel's spaceTimePredictorNonlinear function aderPicardLoopNonlinear where the amount of Picard iterations is determined.

5.2 C++ Chrono Library

Because the task-offloading approach depends on MPI_THREAD_MULTIPLE which is currently not supported by Score-P, a manual instrumentation is needed. For this the C++ Chrono library is employed that includes the clock with the shortest tick period available[21].

Algorithm 3 C++ Chrono instrumentation

```
1:  function FOO
2:      rank_id ← getRankId()
3:      thread_id ← getThreadId
4:      start ← time()
5:      picardIterations ← solverpredictionAndVolumeIntegralBody()
6:      stop ← time()
7:      if file exists then
8:          append (stop − start) and picardIterations to result file
9:      else
10:         create result file with name depending on rank_id & thread_id
11:         append (stop − start) and picardIterations to result file
```

Instrumentation with the C++ Chrono library is quite intuitive as it is only necessary to compare time-stamps and store the result in a file whose name indicates to which rank and thread it belongs. For every visit of the instrumented region the measured time is appended. In addition to the already with Score-P instrumented regions the run() function from the MigratablePredictionJob class is instrumented to trace the wall-time. Furthermore, a more granular insight on Picard iterations is granted by differentiating whether a MigratablePredictionJob is remote or from its original rank by instrumenting in the classes' function handleLocalExecution() and for remote jobs in handleExecution(). Finally, for experiments with the FV limiter, the run() method from the class LimitingADERDGSolver::FusedTimeStepJob is recorded.

5.3 Intel Trace Analyzer

The Intel Trace Analyzer is a tool to analyze performance and load balancing of MPI applications. Similar to Score-P the user has to instrument their regions of interest. However, this tool will also collect information of communication patterns and much more[22]. It was used to get an insight on the function behaviour of the CCZ4 scenario with a limiter. As the code for instrumentation with the trace analyzer was already

implemented, it was only necessary to activate it by exporting a compile flag. Several regions including functions of the limiter and ADER-DG solver are examined.

6 Experiments

In this chapter all experiments are presented and the are results discussed. All experiments are conducted on the CoolMUC-2 Linux Cluster. Combined, it has 384 nodes with 28 cores per node which all clock at 2.6GHz[23]. Every code was compiled with Intel MPI 2018, for shared memory Intel TBB 2018 was used. Additionally GSL2.5 was deployed for CCZ4 and limiter experiments. All metrics (time,visits) were gathered using Score-P, the C++ Chrono library or Intel Trace Analyzer . Wall-time bias by instrumentation was inspected and found to be negligible (below 1%).
The following table shows the default values for the computational domain size and offset for the scenarios.

scenario/computational domain	size	offset
Euler	[1.0;1.0;1.0]	[0.0;0.0;0.0]
Gauge	[1.0;1.0;1.0]	[0.0;0.0;0.0]
CCZ4	[80.0;80.0;80.0]	[-40.0;-40.0;-40.0]
CCZ4-Limiter	[40.0;40.0;40.0]	[-20.0;-20.0;-20.0]

Table 6.1: Default computational domain attributes for the scenarios

In the next table the default grid size for a given node setup and presence/absence of outside cells can be seen.

nodes/grid	no outside cells	one outside cell on each side
2 nodes	27x27x27	25x25x25
28 nodes	81x81x81	no experiments

Table 6.2: Default grid size for number of nodes

Unless explicitly stated, the grid size and the computational domain will not be different in the following experiments. The metric for load imbalance is explained in chapter 4. For Euler the entropywave scenario was employed, but for simplicity it will be referred to as Euler or the Euler scenario.

6.1 Imbalanced Domain

The following comparison is based on the three scenarios Euler, CCZ4 and Gauge all with a ADER-DG solver run on two computing nodes. It shall give an insight of the impact of outside cells on load imbalances. First, the Euler scenario with order nine and 200 time steps. Second, the Gauge scenario with order three and 20 time steps and finally CCZ4 also with order three and also 20 time steps yields the output below.

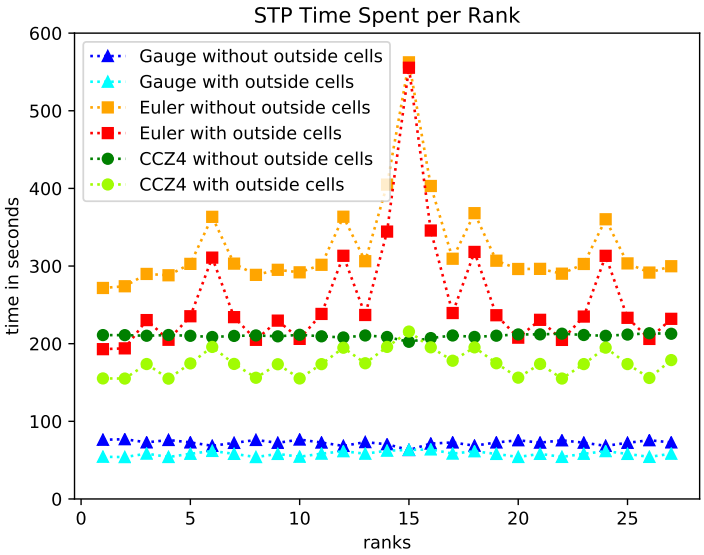


Figure 6.1: Impact of outside cells on the scenarios

For all scenarios one run with zero and another run with one outside cell on each side is plotted. Apparently, Euler is the most imbalanced scenario for both cases of outside cells with an imbalance value of 1.74 for no outside cells compared to 2.16 for one outside cell. This means that the most heavily loaded rank (15) spends more than two times in STPs than the average rank. It is worthy of mention that the time spent in STPs behaves symmetrical for all ranks with rank 15 as the symmetrical axis which is due to the partitioning of Peano and the scenario. A similar pattern appears for CCZ4 with outside cells set to one, where also rank 15 is the one with the greatest overall time in STPs. Compared to Euler it is less imbalanced as it only measures 1.23. However, without outside cells it yields the most balanced result of all scenarios as the metric is only 1.02. Finally, for Gauge in both cases decently balanced outputs yield. Again without outside cells the run is slightly better distributed than with outside cells set to one (1.07 vs. 1.10). In conclusion, Gauge is the most stable scenario, however CCZ4 is

even better balanced without outside cells.

6.2 Numerical Imbalances

A different cause for imbalances can be found in the numerics. Such a numerical imbalance has its roots in the STP jobs, precisely in the number of Picard iterations. The correlation of STP wall-time per rank and Picard iteration per rank shall be illustrated in the following using the same scenarios as in the previous section with outside cells set to one. Moreover, the order of Gauge and CCZ4 is now set to five.

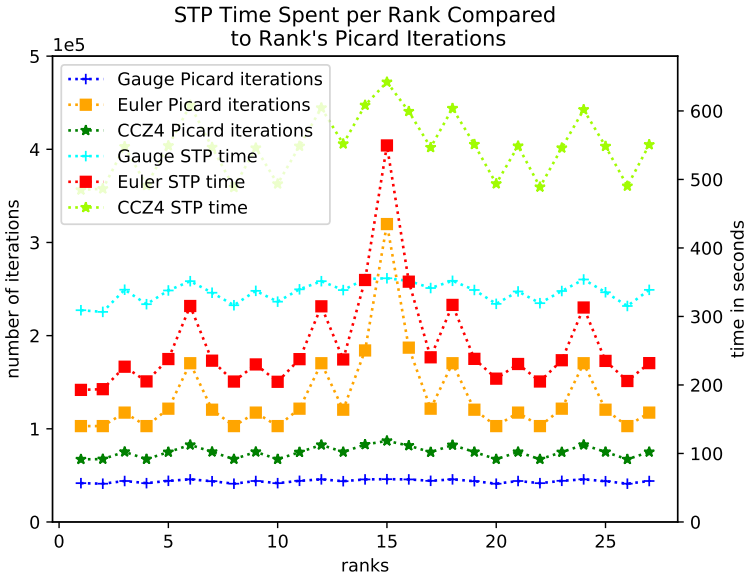
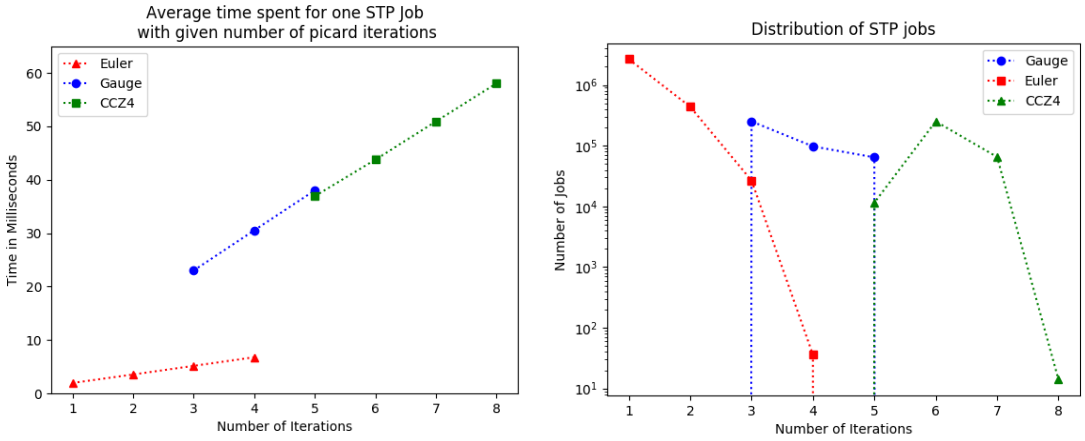


Figure 6.2: Comparison of overall STP time per rank to Picard iterations for each scenario

Clearly, the lines of the number of Picard iterations and the time spent in STPs correlate for all scenarios with one exception to be the rank 15 of the Gauge scenario which has a slightly higher STP time but lower number of Picard iterations. Important to note is that Euler has fewer Picard iterations for an equal grid and amount of time-steps than CCZ4 and Gauge. Yet the overall time spent in STPs per rank is relatively lower than in the two scenarios. This raises the question of how expensive one STP task is which is illustrated on the next page in 6.3.



(a) Average time costs of a STP task

(b) Number of jobs

Figure 6.3: Costs of a STP job with different numbers of Picard iterations for each scenario

Interestingly, the costs for a STP job increases linearly with the number of Picard iterations, as seen in figure 6.3a. Noteworthy is that Euler is the only scenario with STP tasks that do only have one Picard iteration. Furthermore, the slope for the cost line of Gauge and CCZ4 is seemingly the same while the gradient for Euler is lower in comparison. The reason for that is the lower number of variables in Q for Euler. When looking at the distribution of the types of STP jobs with a certain number of Picard iterations, which is illustrated in 6.3b, the most jobs in the Euler scenario have one iteration. Another observation is that the costs for one STP job marginally depends on the rank, for example in the Euler scenario a task with one iteration on rank one costs 1.9 milliseconds while on rank four the average cost is two milliseconds.
A different cause for numerical imbalances in ExaHyPE is the limiter with the FV solver. CCZ4 on 28 nodes with a grid of 81x81x81 was used to illustrate the impact of a limiter on load imbalances. The pie-charts of figure 6.4 on the following page show the distribution of time spent in certain groups of functions of each thread. Data was collected and visualized with Intel Trace Analyzer. Noticeable is that the functions in the ADER-DG solver group seem well balanced where in contrast the group of the limiter is not equally distributed. Furthermore, the red group represents MPI functions which also include wait-times. It can be easily seen that all threads with a lower proportion of the limiter group have a higher share of the MPI group which means that they have a greater wait-time.

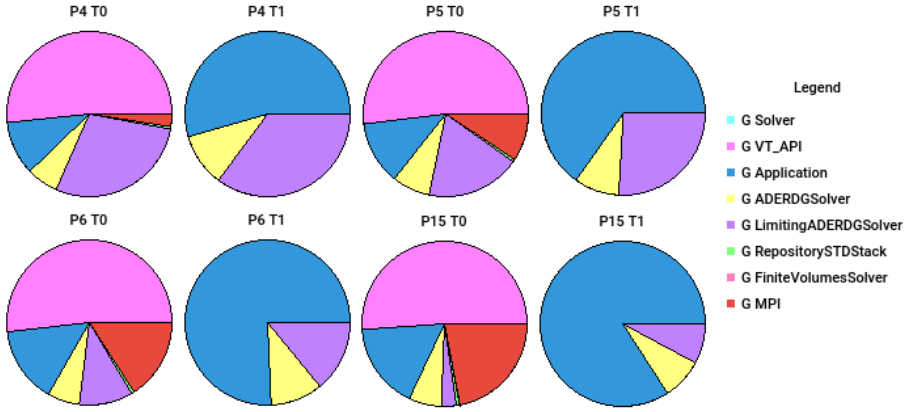


Figure 6.4: Load balance pie charts taken from Intel Trace Analyzer

The following figure 6.5 indicates the imbalanced distribution of the FTS jobs of the limiter. Clearly, the STP jobs are well balanced among all ranks just as already observed in 6.1 plot for CCZ4 without outside cells. Definitely, the cause for wait-times are the FTS jobs which have an imbalance of 1.48. This means that the rank with the highest time spent in FTS takes almost one and a half time longer than the average rank.

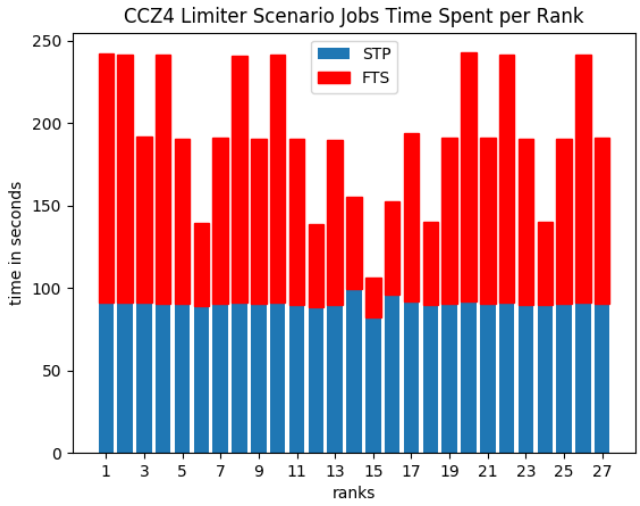
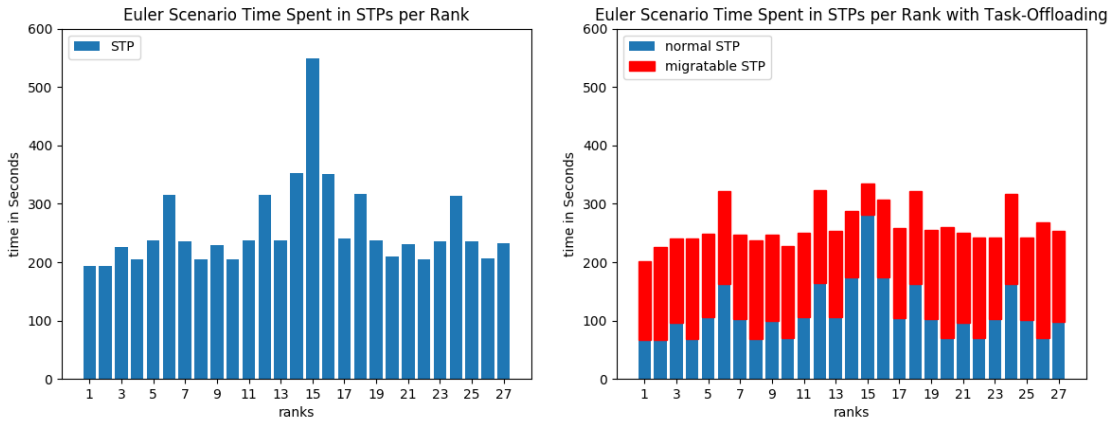


Figure 6.5: Load imbalance of the CCZ4 scenario with a limiter

6.3 Task-Offloading Approach

In this section the task-offloading approach introduced in chapter 4 shall be thoroughly tested on whether an improvement in terms of total elapsed wall-time and load balancing for the ranks can be seen. Because the Gauge scenario is already quite balanced and for the CCZ4 with limiter scenario the FTS jobs are causing wait-times, which are not considered for offloading, only Euler and the CCZ4 with no limiter will be consulted in the following. For both scenarios the computational domain has one outside cell on all sides. Beginning with Euler with the same setup as before and two cores per rank, task-offloading is deployed and evaluated. For this run with 200 time-steps the following result unfolds.



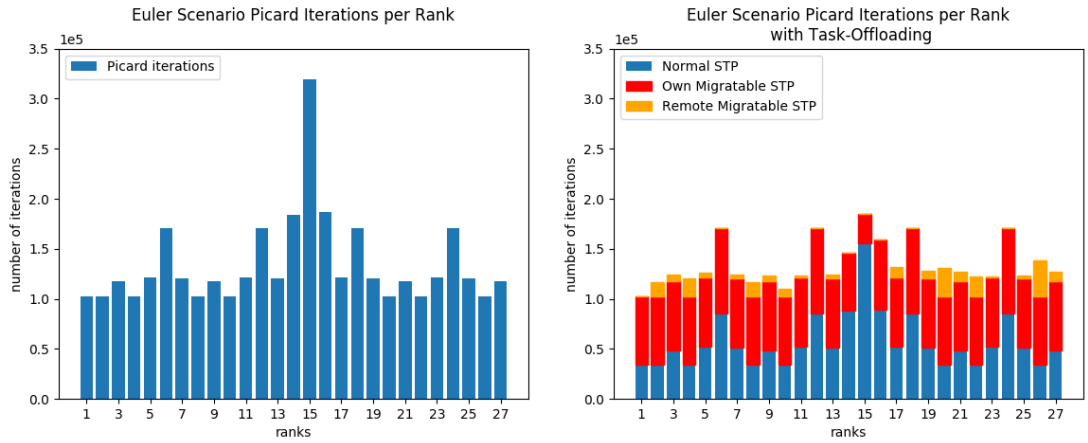
(a) STP time per rank normal run

(b) STP time per rank task-offloading

Figure 6.6: Comparing STP times for Euler scenario with and without task-offloading

Compared to a normal run, a meaningful difference in terms of imbalances and total elapsed time can be seen. While the normal run lasted 297.9 seconds, with Task-Offloading the wall-time was 196.34 seconds. The imbalance is 2.16 vs. 1.37. In figure 6.6a rank 15 is the most heavy loaded in terms of STP time, whereas the ranks in figure 6.6b are clearly more balanced. This is because rank 15 almost has offloaded almost all of its migratable (red) tasks and only keeps its non-offloadable (blue) normal STP jobs. The figure 6.7 on the next page also grants insights on how the Migratable/PredictionJobs are handled in terms of Picard iterations. All ranks with a high number of Picard iterations, obviously, do not receive tasks from other ranks. Thus, these ranks only run iterations for their own MigratbablePredictionJobs or standard PredictionJobs (red and blue). Other ranks execute tasks from remote jobs

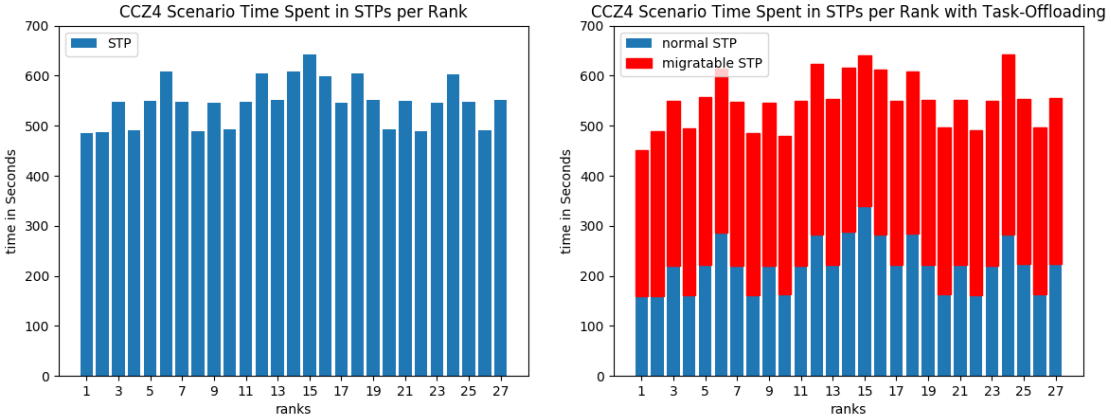
(orange). They make up to 4.4% of all Picard iterations.



(a) Picard iterations per rank normal run (b) Picard iterations per rank task-offloading

Figure 6.7: Comparing Picard iterations with and without task-offloading

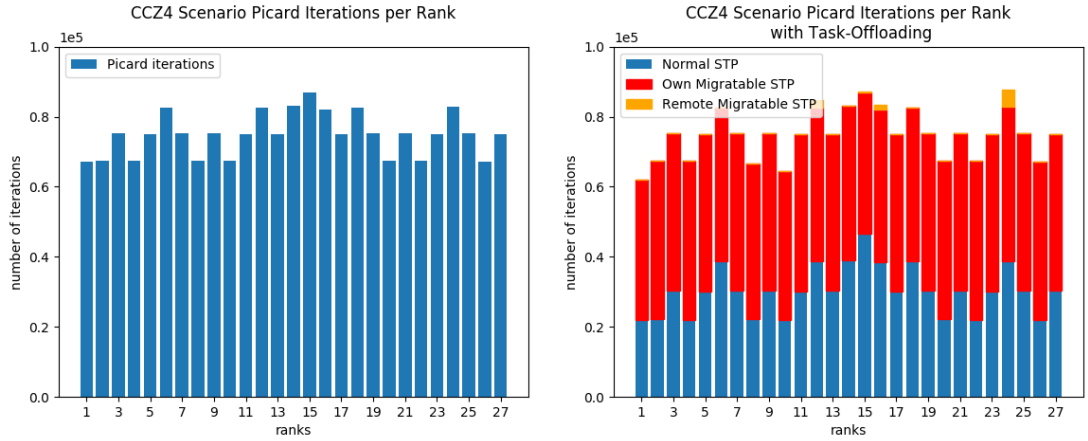
Next on, CCZ4 was tested with 20 time-steps on order five. Again, with task-offloading enabled a lower overall wall-time can be achieved. However, the difference is not as big as with the Euler scenario. For CCZ4 the normal run lasted 720 seconds whereas with task-offloading only 700 seconds elapsed in the best case.



(a) STP time per rank normal run (b) STP time per rank task-offloading

Figure 6.8: Comparing STP times for CCZ4 scenario with and without task-offloading

Similar to Euler the run is better balanced(1.17 vs 1.23). However, only 0.4% of all Picard iterations are from remote jobs, as seen in Figure 6.9. Noteworthy is that the performance of the task-offloading approach may vary and therefore can yield different run-times, since the offloading is reactive as mentioned chapter 4 which does not guarantee deterministic behaviour. For five runs the average elapsed time was 720 seconds with a maximum of 732 seconds.



(a) Comparing Picard iterations normal run

(b) STP time per rank task-offloading

Figure 6.9: Comparing Picard iterations with and without task-offloading

When testing the offloading_progress and spawn_update_as_background_thread settings with the Euler scenario for order nine, the following result yields for 200 time steps.

offloading_progress/ spawn_update_as_background_thread	overall wall-time	imbalance
none/true	183.1s	1.25
none/false	184.79s	1.26
progress_task/true	202.23s	1.23
progress_task/false	196.6s	1.23

Table 6.3: Comparing different settings for task-offloading & optimisation

Consistently with offloading a lower total run wall-time is achieved. Notably, progress_task with spawn_update_as_background_thread set to true scores the lowest imbalance, but has a higher total run-time.

Another experiment was conducted where the order of the polynomial was changed,

ceteris paribus.

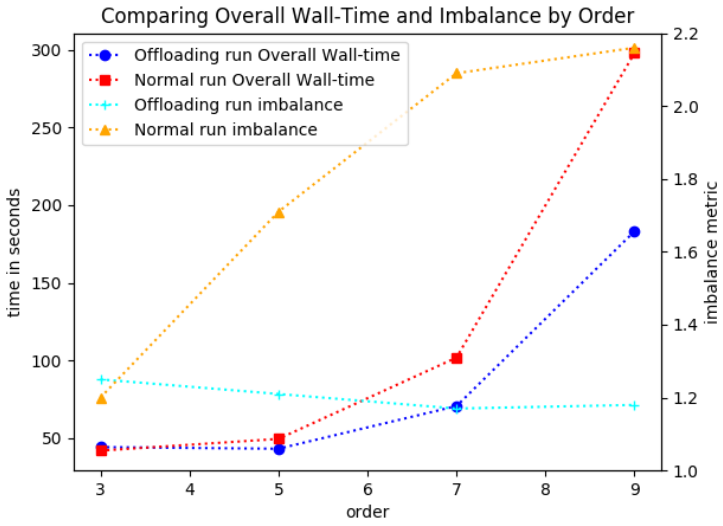


Figure 6.10: Impact of polynomial order with task-offloading for the Euler scenario

Figure 6.10 shows that while for order three the result is not significantly different, deploying task-offloading outperforms a normal run with no run-time load balancing. Generalizing, the higher the order, the better is the result. Analogously, the overall elapsed time increases with an increase of the order for a normal run. In contrast to that, task-offloading has the lowest imbalance with order 7, but it changes only marginally for different orders.
Finally, the performance of the task-offloading approach shall be tested with a huge scenario of Euler with a grid of 81x81x81 with order nine, 28 computing nodes were deployed. In Figure 6.11 on the following page the difference of using a specific number of cores per rank is illustrated. For task-offloading the lowest elapsed time is achieved with 24 cores per rank. As seen as before with two cores per rank, it is faster than a default run. With four cores per rank the overall wall-time is still lower but for more cores per rank the normal run is faster in comparison to the task-offloading approach. Notably, its fastest run is with 28 cores per rank. The reason behind the performance drop with more cores per rank with task-offloading in comparison to a normal run is due to a higher communication overhead. When looking on load distribution, it is still more balanced than a normal run but it fails to achieve a lower overall wall-time.

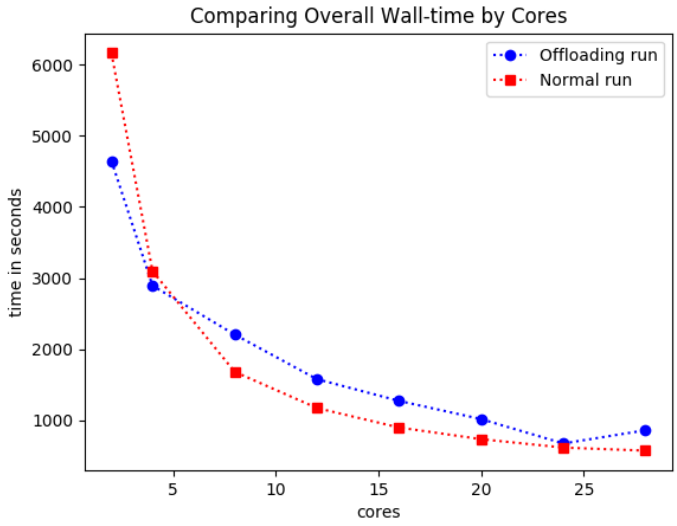


Figure 6.11: Comparison of different numbers of cores per rank with Euler

When looking deeper into how many STP tasks are offloaded per time-step the following plot results.

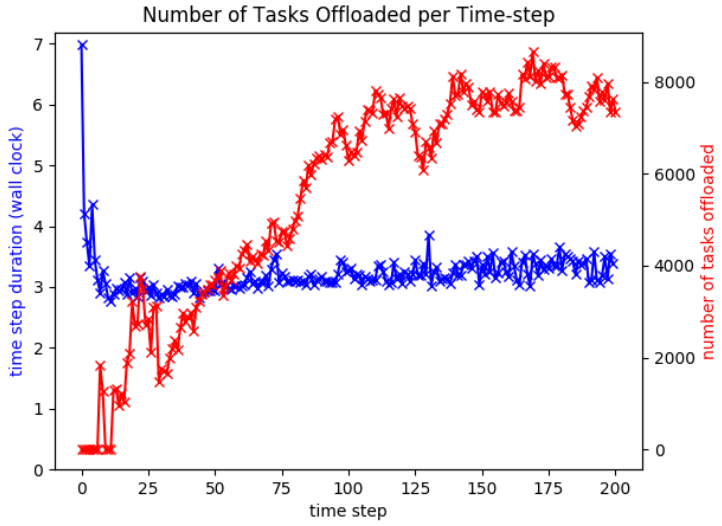


Figure 6.12: Number of tasks offloaded and time-step duration

Figure 6.12 also shows how the elapsed time for one step changes over the course of the whole run. This was instrumented from the previous Euler scenario with 24 cores per rank. In the beginning the time step duration (blue) is at its maximum but after a few steps it already converges. The number of tasks offloaded (red) increases over the time-line. Another observation is that whenever the number of tasks offloaded rises analogously with an increase of the time step duration, a drop in the number of task can be seen for the next time-step. This is due to the reactive behaviour of the task-offloading approach.

6.4 AMPI/Charm++

Because of an inscrutable bug in AMPI it was not possible to successfully run ExaHyPE when it was compiled with AMPI. The application terminated successfully only when one rank was used. However, for one rank no load imbalances can be measured. It was possible to locate a point where the application seems to get stuck in a loop of Peano's message send-receive pool. However, this does not have to be the error. The team working on AMPI/Charm++ is notified and look into this issue. Multiple networking layers for AMPI were tested such as MPI, but all ran into a failure.

7 Conclusion and Future Work

In this thesis, several causes for load imbalances were examined and found to have a huge impact on overall run-time for modern numerical software. On one hand, an unequal distribution of work to ranks by an imbalanced computational domain caused a load imbalances difference of 0.42 for the Euler scenario, which also means that the overall wait-time is increased. On the other hand, numerical imbalances in terms of Picard iterations in the space-time prediction-jobs were found to be the cause for load imbalances, as the number of iterations correlate with the wall-time in STPs. Tackling these with the task-offloading approach is most of the time of avail, since a reduction of the total elapsed time up to a third was measured. In terms of Picard iterations it was shown how the tasks from the critical rank were distributed to other ranks which otherwise would be idle. The behaviour of the approach was also tested for different polynomial orders of the solver and found to outperform a run without load balancing for higher orders. Furthermore, the costs of STPs were examined whereas the Euler scenario yielded the fewest [1,..,4] number of Picard iterations per STP task and CCZ4 the most [5,..,8]. Because the CCZ4 scenario with task-offloading could not show a real improvement in reducing overall wall-time, it is still necessary to have a deeper look into the cause. For example an interesting information to gather would be why so few tasks are offloaded in comparison to the Euler scenario. Furthermore, data to show the communication costs for task-offloading has to be gathered to get a more granular look on how the approach behaves with a higher number of cores per rank, as it tends to have a slightly higher overall wall-time than a normal run with no run-time load balancing. Finally, instrumentation with AMPI/Charm++ still has to be done, when the bug which causes ExaHyPE to crash is fixed. It would be interesting to see if this approach can also show improvements in terms of reducing total wall-time. A comparison of the two ways of run-time load balancing for the different scenarios and test cases could show where one of the approaches might have an advantage in comparison to the other.

List of Figures

List of Tables

Bibliography

[1] M. B. Thomas Sterling Matthew Anderson, *High Performance Computing Modern Systems and Practices*. Morgan Kaufmann, 2018, ISBN: 978-0-12-420158-3.

[2] J. Charles, P. Jassi, N. Ananth, A. Sadat, and A. Fedorova, "Evaluation of the intel® core™ i7 turbo boost feature," *IEEE Xplore*, pp. 188–197, 2009. DOI: 10.1109/IISWC.2009.5306782.

[3] W. Kendall. (last visited 12-03-2020). Mpi send and receive, [Online]. Available: https://mpitutorial.com/tutorials/mpi-send-and-receive/.

[4] M. P. I. Forum. (last visited 27-02-2020). Mpi: A message-passing interface standard, version 3.0, [Online]. Available: https://www.mpi-forum.org/docs/mpi-3.0/mpi30-report-book.pdf.

[5] J. Reinders, *Intel Threading Building Blocks Outfitting C++ for Multi-Core Processor Parallelism*. O'Reilly Media, Inc., 2007, ISBN: 978-0-596-51480-8.

[6] A. Reinarz, D. E. Charrier, M. Bader, L. Bovard, M. Dumbser, K. Duru, F. Fambri, A.-A. Gabriel, J.-M. Gallard, S. Koppel, LukasKrenz, L. Rannabauer, L. Rezzolla, P. Samfass, M. Tavelli, and T. Weinzierl, "Exahype: An engine for parallel dynamically adaptive simulations of wave problems," pp. 1–20, 2019. DOI: arXiv:1905.07987v1.

[7] ParaView. (last visited 10-03-2020). Paraview, [Online]. Available: https://www.paraview.org/.

[8] T. Weinzierl, "The peano software—parallel, automaton-based, dynamically adaptive grid traversals," *ACM Trans. Math. Softw.*, vol. 45, no. 2, pp. 2–6, 2019. DOI: http://dx.doi.org/10.1145/3319797.

[9] T. W. D.E. Charrier, "Stop talking to me–a communication-avoiding ader-dg realisation," pp. 4–6, 2018. DOI: arXiv:1801.08682v1.

[10] M. Dumbser, O. Zanotti, R. L. ere, and S. Diot, "A posteriori subcell limiting of the discontinuous galerkin finite element method for hyperbolic conservation laws," *Computational Physics*, p. 2, 2015. DOI: arXiv:1406.7416v3.

[11] I. Peshkov, W. Boscheri, R. Loub'ere, E. Romenski, and M. Dumbser, "Theoretical and numerical comparison of hyperelastic and hypoelastic formulations for eulerian non-linear elastoplasticity," *Noname*, pp. 1–2, 2018. DOI: arXiv:1806.00706v1.

[12] L. Bovard and S. Koeppel. (last visited 04-03-2020). Ccz4: Gauge wave benchmark, [Online]. Available: http://dev.exahype.eu/astrophysics-userguide/dirhtml/Benchmarks/gaugewave/.

[13] M. Dumbser, F. Guercilena, S. Köppel, L. Rezzolla, and O. Zanotti, "A strongly hyperbolic first-order ccz4 formulation of the einstein equations and its solution with discontinuous galerkin schemes," pp. 1–3, 2017. DOI: arXiv:1707.09910v1.

[14] G. Z. Laxmikant V. Kale, "The charm++ programming model," pp. 1–6, 2013.

[15] UIUC-PPL. (last visited 25-02-2020). 3. adaptive mpi (ampi), [Online]. Available: https://charm.readthedocs.io/en/latest/ampi/manual.html.

[16] L. V. K. Sam White, "Optimizing point-to-point communication between adaptive mpi endpoints in shared memory," pp. 1–5, 2017.

[17] P. Samfass, T. Weinzierl, D. E. Charrier, and M. Bader, "Tasks unlimited: Lightweight task offloading exploiting mpi wait times for parallel adaptive mesh refinement," p. 2, 2019. DOI: arXiv:1909.06096v1.

[18] T. W. D.E. Charrier B. Hazelwood, "Enclave tasking for discontinuous galerkin methods on dynamically adaptive meshes," 2020. DOI: arXiv:1806.07984v3.

[19] A. K. C. Roessel, D. an Mey, S. Biersdorf, K. Diethelm, D. Eschweiler, M. Geimer, M. Gerndt, D. Lorenz, A. Malony, W. E. Nagel, Y. Oleynik, P. Philippen, P. Saviankou, D. Schmidl, S. Shende, R. Tschuter, M. Wagner, B. Wesarg, and. F. Wolfr, "Score-p – a joint performance measurement run-time infrastructure for periscope, scalasca, tau, and vampir," pp. 2–4,

[20] Score-P. (18-02-2020). Changelog, [Online]. Available: https://www.vi-hps.org/cms/upload/packages/scorep/scorep-5.0-ChangeLog.txt.

[21] C. preference website. (last visited 20-01-2020). Date and time utilities, [Online]. Available: https://en.cppreference.com/w/cpp/chrono.

[22] I. Corporation. (last visited 05-02-2020). Intel trace analyzer and collector, [Online]. Available: https://software.intel.com/en-us/trace-analyzer.

[23] LRZ. (last visited 03-04-2020). Coolmuc-2, [Online]. Available: https://doku.lrz.de/display/PUBLIC/CoolMUC-2.