universität wien

# MASTERARBEIT / MASTER'S THESIS

Titel der Masterarbeit / Title of the Master's Thesis

## „Coupling of Applications for Progress-Driven Co-Scheduling in the Open Community Runtime“


verfasst von / submitted by

Johannes Ender


angestrebter akademischer Grad / in partial fulfilment of the requirements for the degree of

Master of Science (MSc)

Wien, 2020 / Vienna, 2020

| | |
|---|---|
| Studienkennzahl lt. Studienblatt / degree programme code as it appears on the student record sheet: | UA 066 910 |
| Studienrichtung lt. Studienblatt / degree programme as it appears on the student record sheet: | Masterstudium Computational Science |
| Betreut von / Supervisor: | Univ.-Prof. Dipl.-Ing. Dr. Siegfried Benkner |
| Mitbetreut von / Co-Supervisor: | Dr. Jiri Dokulil |

**Abstract**

In this thesis methods for progress-based co-scheduling of coupled applications will be investigated using the *Open Community Runtime (OCR)* programming model. A recently introduced coupling mechanism, which enables the OCR to automatically balance the computational resources of two coupled, concurrently running instances will be used. As not all algorithms show regular computational load, it is possible that simulation and analysis applications with different progress rates drift apart. For achieving high performance, the computational resources should be dynamically adjusted to align progress of simulation and analysis. Examples for such irregular computational workloads from scientific applications are adaptive mesh refinement or N-body simulations. A simulation and analysis application will be selected and - if necessary - ported to the OCR programming model. These applications will then be used to perform different experiments that are being discussed and compared to to the traditional post-processing workflow.

**Zusammenfassung**

In dieser Arbeit werden Methoden zum fortschrittsbasierten Co-Scheduling von gekoppelten Anwendungen unter Verwendung des *Open Community Runtime (OCR)* Programmiermodells untersucht. Es wird ein vor Kurzem vorgestellter Kopplungsmechanismus verwendet, der es der OCR ermöglicht, die Rechenressourcen von zwei gekoppelten, gleichzeitig laufenden Instanzen automatisch auszugleichen. Da nicht alle Algorithmen eine reguläre Rechenlast aufweisen, ist es möglich, dass Simulations- und Analyseanwendungen mit unterschiedlichen Fortschrittsraten auseinanderdriften. Um eine hohe Leistung zu erzielen, sollten die Rechenressourcen dynamisch angepasst werden, sodass Fortschritt von Simulation und Analyse aufeinander abgestimmt werden. Beispiele für solche irregulären Rechenlasten aus wissenschaftlichen Anwendungen sind adaptive Gitterverfeinerungen oder Mehrkörpersimulationen. Eine Simulations- und Analyseanwendung wird ausgewählt und - falls erforderlich - auf das OCR-Programmiermodell portiert. Diese Anwendungen werden dann verwendet, um verschiedene Experimente durchzuführen, die diskutiert und mit dem traditionellen Postprocessing-Arbeitsablauf verglichen werden.

# Contents

# List of Figures

# List of Tables

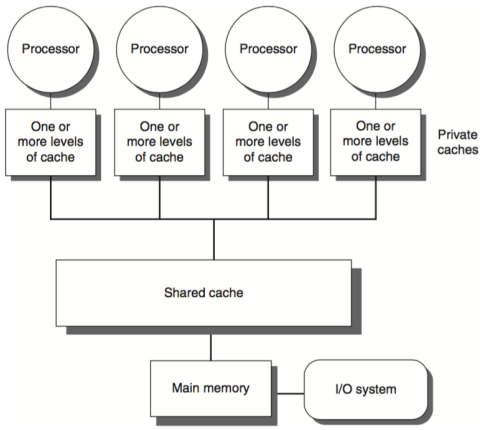
# Chapter 1

# Introduction

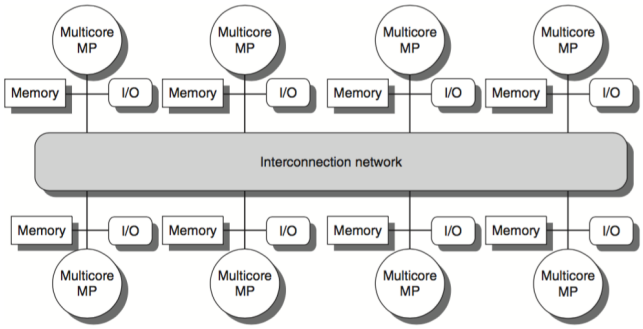
Nowadays, a scientific world without the aid of computers is hardly imaginable. All kinds of sciences rely on the computational power of (super-)computers. They are for example used to analyze the (human) DNA or to perform numerical weather prediction.
Let's take molecular dynamics simulation as an examplary scientific application. In molecular dynamics simulations a potential energy landscape is calculated which is used to calculate the forces acting upon every single atom and by using Newton's equations of motion to determine the trajectory of the atom's motion. The computational load grows with $\sim O(N^2)$, so if one wants to simulate bigger molecules, computational resources can quickly become a limiting factor.

For a long time Moore's law governed the evolution of processing power of CPUs and ensured a steady increase in performance, but increasing the frequency of CPUs was slowly going to be not enough anymore. This is where concurrency came into play. New CPU architectures incorporating parallel executing processing units made sure that the trend in computational processing power continued.
But whereas performance benefits due to increasing CPU frequencies came for free - at least from an application developer's point of view - this is not the case for the new architectures anymore. Making use of parallel hardware and exploiting parallelism of a program is being done on multiple levels of abstraction. So-called instruction-level-parallelism subsumes a set of hardware techniques for parallelizing basic CPU instruction execution and is nowadays used by every processor.
Then the compiler too can - to a certain extent - extract parallelism from source code. But probably the most important level for identifying potentially parallelizable parts of a program is at the programming level. Thus, to make full or better use of these architectural changes, the task of application development gets more involved.

Many programming models have been developed to assist application developers with their job. Here, when talking about theses programming models, an important distinction concerning the properties of target machines has to be made. The multiple, parallel compute nodes either have access to the same memory or they don't. Usually, this is referred to as shared memory and distributed memory systems (see Figures 1.1a and 1.1b). In the shared memory multicore system, the single processor cores have dedicated, private caches as well as - in most cases - a shared last level cache and shared main memory. In the distributed memory system compute nodes are separated and do not share any memory with each other. Of course, combinations of these systems exist and interconnected shared memory compute nodes can form a distributed memory system. Large clusters would be an example for such a machine.




(a) Shared Memory




(b) Distributed Memory

Figure 1.1: Memory architectures (Hennessy and Patterson, 2011, p. 347 f.)

Different programming models exist to exploit either of these - or even both - types of system. (Diaz et al., 2012) discusses the most-widely used programming models for shared and distributed memory systems. Two of the most prominent parallel program-

ming models are *OpenMP* [1] and the *Message Passing Interface (MPI).*[2]
OpenMP is used for shared memory parallelization. It uses a pool of threads that execute regions of code in parallel that have been marked with special compiler directives. After this parallel region a single main thread continues work in a sequential fashion.
As its name already suggests, MPI defines an interface for the exchange of messages between compute nodes of a distributed-memory system as a means for communication. Often the compute nodes operate in a *Single Program Multiple Data (SPMD)* fashion, which means that the same program is executed on different nodes on different data. OpenMP and MPI often are used in a hybrid programming approach in which one uses shared and distributed memory parallelization.

With leading Top500 supercomputers having passed the petascale[3] mark some years ago, many national and international projects have been started to pave the way for exascale computing. Developments of the last years brought a shift towards more heterogeneous architectures with it. Previously having consisted only of CPUs as computing devices in the past, nowadays one can find GPUs, FPGAs, manycore coprocessors, DSPs and more as parts of the machines. It is assumed that exascale systems exhibit even more heterogeneity in their hardware. Deeper memory hierarchies and CPUs combined with GPUs and other accelerators lead to a decreased mean time between failure.(Moreland, 2012, p. 6)
The bulk synchronous parallel methods like MPI mostly in use nowadays are not suitable for such systems, as work is mapped to the compute nodes statically. In case a of a compute node failure, application execution cannot recover by dynamically moving this nodes tasks to a different one and execution has to be terminated.
A more flexible approach are task-based runtime systems that will be introduced in chapter 2.2. These runtimes allow application developers to express an application as a set of interacting tasks that are executed asynchronously. Scheduling this application and distributing data on shared or distributed memory machines is handled by the runtime.

[1] https://www.openmp.org/
[2] https://en.wikipedia.org/wiki/Message_Passing_Interface
[3] 1 petaflop $= 10^{15}$ floating point operations per second

## 1.1 Motivation

As mentioned in the previous chapter, the computational power of supercomputers is steadily increasing. Scientists make use of this development by running more and/or longer simulations. These simulations in turn also generate more results that subsequently need to be analysed. Now the "traditional" approach used to perform these two tasks is by doing them sequentially: run the simulation, fetch the results and run some analysis/visualization on the data. The problem with this post-processing approach is, that potentially very large amounts of simulation data need to be transferred from the supercomputer that ran the simulation to the scientists computer. This either takes an unbearably long time or is just not possible at all.
A remedy for this unpleasent situation would be to already analyse the data on some visualization machine that could also reside in the same cluster as the machine running the simulation (red arrows in Figure 1.2). Still, the same amount of storage space is being used and the overhead of writing the simulation results to disk and having to read them again by the visualization machine extends the time until the analysis reusults are ready.(Yu et al., 2010, p. 45 ff.)

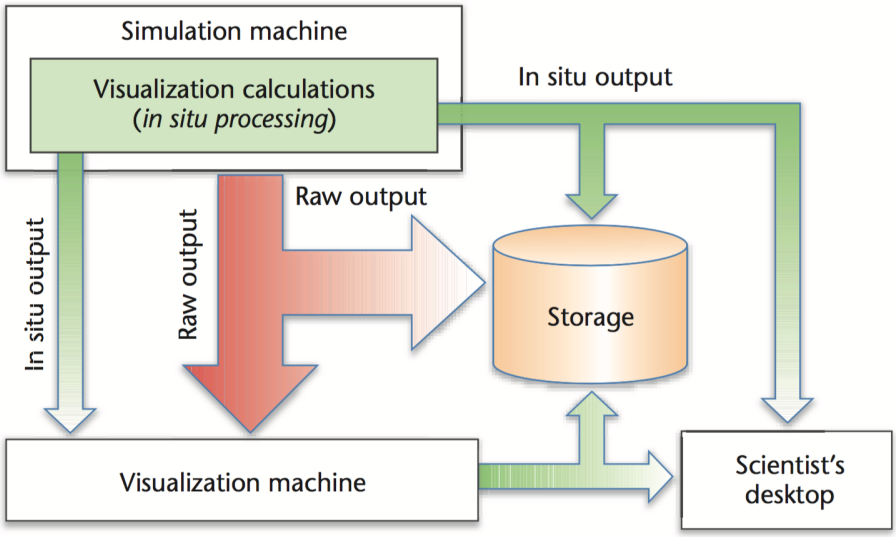



Figure 1.2: Post-, Co- and In-situ processing (Yu et al., 2010, p. 46)

In situ[4] analysis solves this problem (green arrows in Figure 1.2). The analysis part is directly coupled to the simulation and the detour over the file system is bypassed.
But in situ processing is not only more efficient for transferring the simulation data, with certain frameworks it can also be used for simulation steering. That means that, as the

[4]latin for "on site" or "in position" (https://en.wikipedia.org/wiki/In_situ)

simulation is being run, scientist can look at the results being generated and interact with the running simulation and change certain parameters.

Directly coupling the simulation with the analysis application also brings new challenges. To prevent duplication of data, the two applications have to work with the same data structures. The distribution of data is of importance for the performance of the analysis part. If data needs to be moved around between compute nodes, any benefit from going from a post- or coprocessing approach to an in situ approach could be reduced or eliminated.

These beneficial properties of in situ processing are being examined in this thesis. Using a version of the *Open Community Runtime* developed at the *University of Vienna* in the Research Group Scientific Computing, the influence of executing two data-dependent applications concurrently is being explored. A newly developed mechanism presented by Dokulil and Benkner was used to couple a data-producing and a data-consuming application and dynamically schedule their computational resources (Dokulil and Benkner, 2018). Thus, the progress of the two applications could be kept synchronized and the usage of the available computational resources could be improved. The results are compared to the sequential workflow and to a scenario where both applications are running concurrently with statically allocated computational resources.

# Chapter 2

# Related Work

## 2.1 In Situ Processing

Apparently, although in situ visualization and analysis has its roots already several decades ago, terminology for describing the process of in situ visualization and analysis has not been uniform. Therefore the "In Situ Terminology Project" (InSituTermProj, 2016) started out to establish a common terminology for describing in situ methods. The following six categories - that will also be used to describe the in situ approach used in this work - were defined: (InSituTermProj, 2016, p. 3 ff.)

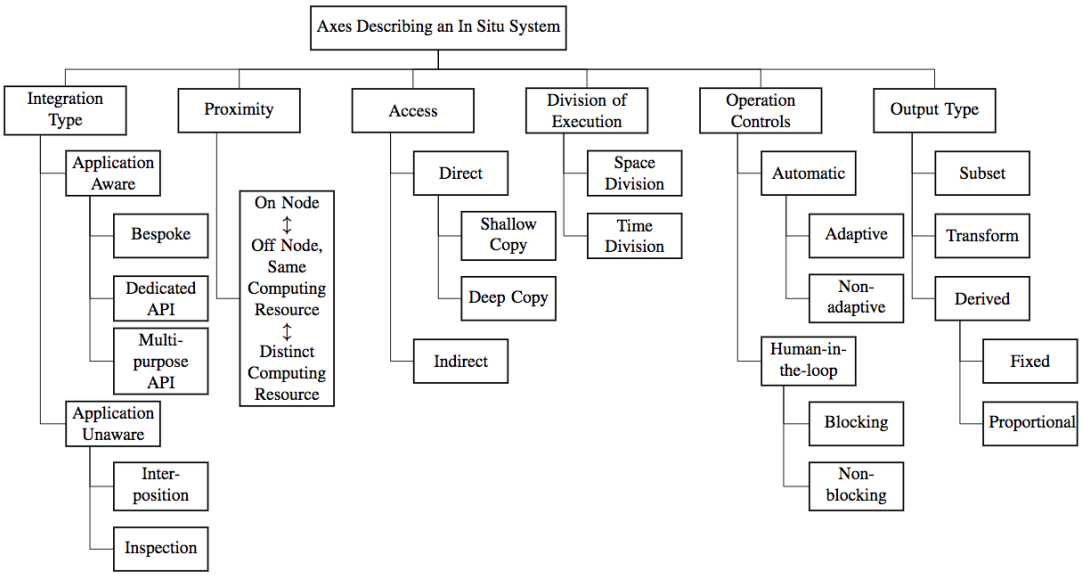



Figure 2.1: In situ axes defined by *In Situ Terminology Project* (InSituTermProj, 2016)

## Integration

The type of integration of the in situ system distinguishes between two approaches. In one approach the simulation is aware of the in situ coupling, i.e. there are library calls or other explicit means to perform in situ analysis. In the other type of integration, the simulation is not aware of any in situ analysis going on. Examples for this situation can be when the simulation uses a certain dynamic library for I/O that in the in situ scenario is being replaced by a library performing the in situ analysis, but has the same interface. Even analysis and pattern deduction from simulation data without the simulation knowing thereof could be imagined. (InSituTermProj, 2016)

## Proximity

Proximity literally is concerned with how close the simulation and the in situ processing are apart from each other. Here, close proximity could mean that both are running on the same compute node or even the same core. Distant proximity could be two processes not even being run within the same cluster, but maybe in different countries or on different continents. Combinations of close and distant proximity can also exist, e.g. two-staged in situ processing, where some pre-filtering is done in close proximity and the filtered data set is forwarded to some visualization machine that is farther away.

## Access

The way data is accessed is closely related to the proximity axis and can easily be confused with it. It distinguishes between approaches where in situ processing operates within the same logical memory space as the simulation does and approaches where they don't. In close proximity scenarios it is most often the case that simulation and analysis operate in the same logical memory space, but running them in two separate process would be an example where this is not the case. The other way around is also possible, i.e. distant proximity but same logical memory space. Running them on separate nodes in a PGAS cluster, for instance.

## Division Of Execution

Execution can be divided either by time or by space. Dividing execution by time means, at any point in time for the duration of running simulation and analysis, only one of them will be running. This does not mean, that the simulation has to finish before the analysis can start, but the two are taking turns. Space division on the other hand divides

the computing resources among the two so they can run concurrently. Clearly, some means of synchronization are needed here, as analysis can progress at most as fast as the simulation does.

## Operation Controls

Some in situ workflows enable the user to decide during runtime what kind of in situ tasks shall be performed. These fall in the *Human-in-the-loop* category, which is further sub-divided into *blocking* and *non-blocking*. In the blocking case the simulation can be paused, in the non-blocking it can't. Besides the Human-in-the-loop category, there also exists the *Automatic* category. Here, the in situ task is fixed before runtime, but nevertheless depending on certain properties during execution can still change it's behaviour slightly.

## Output Type

The procedures applied to simulation data in the in situ processing step are divided into the three sub-categories *Subset, Transform* and *Derived*. If data is being filtered and reduced it is part of the Subset category. Changing the appearance of the data, without reducing the information content is part of the Transform category. Using the original data and creating new data of a different form from it, like rendering an image, is part of the third category, i.e. Derived.

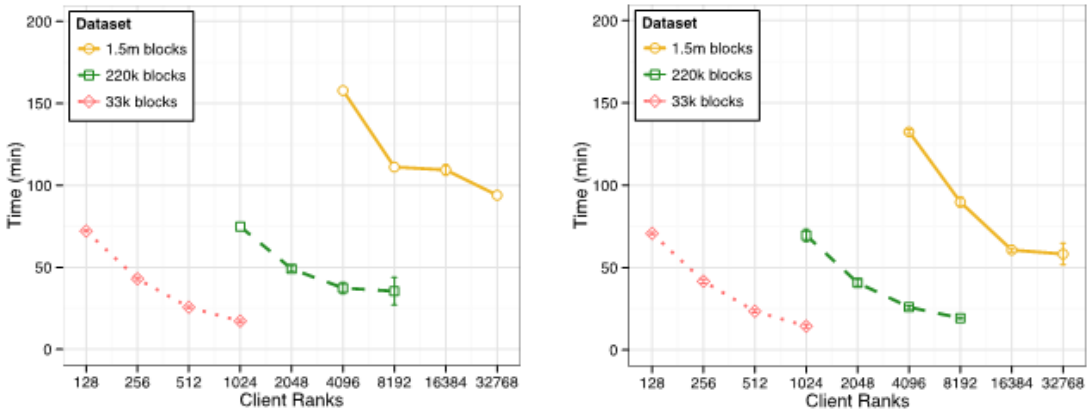
### 2.1.1 Existing In Situ Solutions

In the following, two existing in situ frameworks will be presented.

#### ParaView Catalyst

ParaView is a wide-spread tool used for post-processing visualization tasks. It uses the Visualization Toolkit (VTK)[1] underneath and extends it with a user interface and control logic for parallelization. VTK itself is a software system that can be used for visualization and image processing and defines its own data and execution model.
Now, ParaView Catalyst is a library that directly connects a simulation with ParaView and adds in situ analysis capabilities. To connect to this library one has to write a so-called *adaptor*, which basically has to perform three tasks: *Initialize, CoProcess and Finalize*. These are routines written in the programming language of choice, e.g. C/C++/Fortran/Python, that within their body can make API calls to the Catalyst library.
In the *Initialize* part the library is initialized and certain properties about the simulation can be set. This is done once in the beginning of the simulation.
*CoProcess* is where the better part of the effort needed to create the adaptor goes into. There, the data model from the simulation has to be mapped to the VTK/Catalyst data model. *CoProcess* is called every iteration of the simulation.
The *Finalize* stage cleans up after the simulation and also releases memory that has been acquired. It is called only once per simulation.
A full-blown Catalyst library can be as much as 400 MB, but Catalyst can be reduced to only the functionality that is really needed by the simulation. These library variants are called *Catalyst Editions*. (Ayachit et al., 2015)

[1] https://www.vtk.org/




(a) Classic Workflow

(b) Catalyst Workflow

Figure 2.2: Comparison of ParaView workflows (Bauer et al., 2015, p. 6)

Figure 2.2a and 2.2b show a running time comparison between the classic, post-processing workflow and the in situ workflow using ParaView Catalyst. One can see that the in situ solution performs better than the post-processing solution, but this advantage becomes apparent only starting at a certain data set size. Whereas in the pink curve there is almost no difference in running time, the green curve already shows some improvement and in the yellow curve the running time is almost halved. (Bauer et al., 2015)

### VisIt/LibSim

LibSim is a library developed by scientist working at LLNL[2], CSCS[3] and ORNL[4]. The visualization system it uses is VisIt. The setup that is normally used can be seen in Figure 2.3. A VisIt client running locally on the scientists machine connects to a VisIt server running on a supercomputer whom it tells what data to fetch and what analysis/visualization to perform.

[2]Lawrence Livermore National Laboratory https://www.llnl.gov/
[3]Swiss National Supercomputing Center https://www.cscs.ch/
[4]Oak Ridge National Laboratory https://www.ornl.gov/

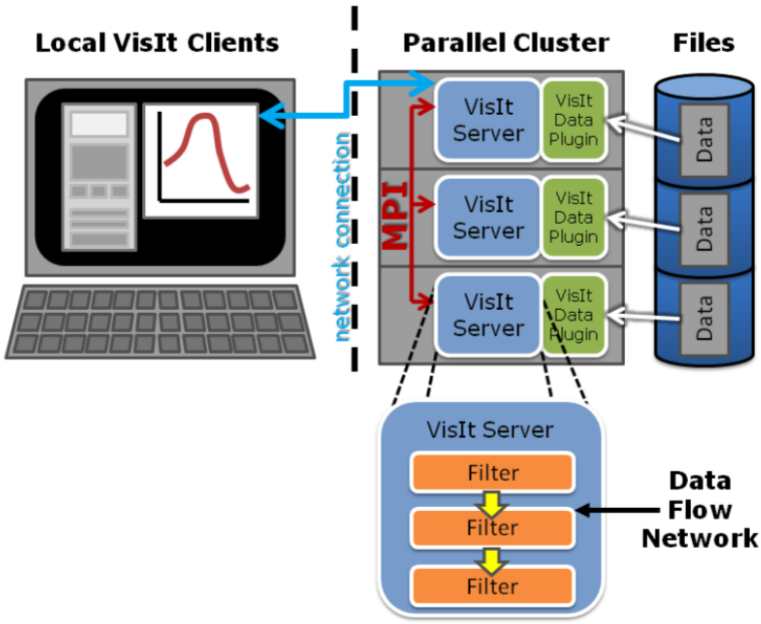



Figure 2.3: Classic VisIt architecture (Whitlock et al., 2011)

The LibSim library can now be used to integrate this VisIt server into a simulation (see Figure 2.4). Similar to ParaView Catalyst, the simulation has to be extended with initialization and finalization steps, but also repeatedly has to check for incoming requests to the server. Unnecessary library loading overhead is avoided by splitting LibSim into two parts: a static front-end library and a dynamic runtime library that is loaded on demand.

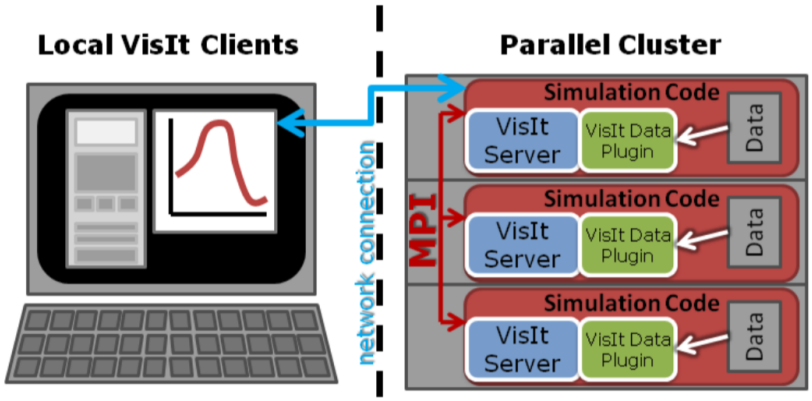



Figure 2.4: VisIt/LibSim architecture (Whitlock et al., 2011)

So, if no in situ analysis is performed/requested at all, the runtime library does not have to be loaded and thus the loading overhead is limited to the front-end library. Results from (Whitlock et al., 2011) show an advantage of an in situ approach compared

to a conventional, file-based approach. There, three scenarios are compared with each other, a simulation generating one single file containing results of all tasks, a simulation where every task generates its own result file and the in situ version, where no intermediate file is generated at all. With the tests being run with different levels of concurrency, in situ always performs better than the single file variant. Compared to the multi-file version, in situ performs equally well in low concurrency runs, but outperforms it when concurrency is increased.

## 2.2 Task-Based Programming Models

In recent years task-based programming approaches have gained popularity. They have the advantage that intricate details of thread programming are abstracted, such that the application developer can concentrate on extracting parallelism from the problem at hand.
All kinds of programming models do support it. OpenMP, a language extension, has introduced tasks in version 3.0. Not only language extensions, but also programming languages themselves introduced task-based parallelism, like C++ `async` that came with the C++11 standard.
Task-based programming models can also come in the form of libraries, as can be seen with the *Threading Building Blocks (TBB)*. And then there is also a plethora of task-based runtime systems, like the Open Community Runtime, Charm++ and many more.

In (Thoman et al., 2017) a taxonomy and a set of features that characterize task-based runtimes is presented, that will be used to compare a selection of such runtimes. The taxonomy can be seen in Figure 2.5.


Figure 2.5: Taxonomy of Many-Task Runtime Systems (Thoman et al., 2017, p. 5)

The two basic **Target Architectures** can - as already mentioned in the previous chapter - be divided in shared and distributed memory architectures. Here, one could even further divide them into homogeneous and heterogeneous

In the past, the **Scheduling Objective** of runtimes has mainly been *execution time*. Nowadays, however, with HPC systems becoming larger and larger, *energy efficiency* has also become an important topic and some runtimes aditionally employ such scheduling objectives. Also *multi-objective* policies have emerged, that try to achieve a compromise between different, conflicting objectives like the two before-mentioned.

**Task Scheduling Methods** are sub-divided into *static, dynamic* and *hybrid* methods. Static scheduling happends before the application is executed, at compilation time and needs some information about the application to perform this task. Examplary properties are the running time or resource usage of a task, the dependencies between tasks or location of the input data. When this data is not available, dynamic or hybrid scheduling is used, depending on how much information about the application is known beforehand. *List scheduling* is an example for a static approach. Here, static lists of tasks are generated that are then worked off at runtime. *Work-stealing* scheduling approaches fall under the dynamic scheduling sub-category. In order to balance the load on the whole system, tasks are stolen from other nodes. Usually these tasks are managed using per-worker

queues.

**Performance Monitoring** is seen as an important part of many-task runtimes, especially for upcoming exascale systems because of their large degree of concurrency. The performance monitoring capabilities are grouped into two sub-categories: **offline** and **online**.

**Offline** monitoring does not use the collected data during runtime of the application, whereas the **online** approach uses it during execution. Offline monitoring is done by the end-user (*Performance Analysis*), while online can also be done by the runtime (*Performance Modelling, Introspection*).

For the **detectability of faults**, three levels of the system are distinguished: distributed execution, process and task (see Figure 2.6).

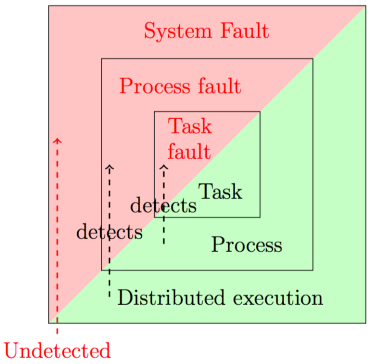



Figure 2.6: Fault Detectability (Thoman et al., 2017, p. 7)

Figure 2.6 shows what kind of faults can be detected by which system. Process and task faults can in certain cases be detected, but with system faults usually no recovery strategy can help in such a situation. (Thoman et al., 2017)

### 2.2.1 Charm++

Charm++ is one of the older representatives of parallel runtimes. It was developed by L. Kale at the University of Illinois, Urbana-Champaign in the early 1990's. Although mainly for distributed memory systems, it can also be run on single- or multicore shared memory machines. It runs on a variety of platforms and infrastructures.
Charm++ is used in several scientific applications, showing it's mature state and production readiness. NAMD is probably one of the most prominent of them.
The objects used by Charm++ to express parallelism are called `chares`. These chares are communicating by calling each others entry methods, which can be seen as some kind of message-driven system. Relationship among chares is organized as a structured dagger (SDAG). Chares are globally identifiable via a unique ID.(Kale and Krishnan, 1993)
A Charm++ program consists of `.cpp`, `.h` and `.ci` files. The `.ci` files contain descriptions of the chare interfaces which are used by the compiler (`charmc`) to generate proxy code that is used locally to marshall information/data and forward the calls to methods of remote chares. (Sterling et al., 2017)
Using the taxonomy presented in the beginning of the chapter, the features of Charm++ can be summarized as in table 2.1.

| | Data Distribution | Scheduling on Shared Memory | Scheduling on Distributed Memory | Performance Monitoring | Fault Tolerance | Target Architecture |
|---|---|---|---|---|---|---|
| Charm++ | i | **m** | **m** | **off/on** | **pf** | **d** |

Table 2.1: Charm++ features
**i**...implicit, **e**...explicit, **m**...multiple (incl. ws), **l**...limited, **ws**...work stealing, **tf**...task faults, **pf**...process faults, **sm**...shared memory, **d**...distributed memory, **on**...online use, **off**...offline use (Thoman et al., 2017, p. 9)

The distribution of the data happens implicitly and multiple scheduling algorithms exist for shared memory systems as well as for distributed memory systems, despite its main target being distributed memory systems. Performance monitoring tools exist for on- and offline usage.

### 2.2.2 StarPU

Development of StarPU started around 2008 at the University of Bordeaux. It is designed to be accelerator-friendly and one of its aims is to reduce porting time of applications from one architecture to a new one. Core parts of the runtime are concerned with task abstraction, data management and scheduling and execution of tasks.

The task abstraction in StarPU is called **codelet**. Codelets are small fractions of an application that, together with their input and output dependencies and other codelets, form an application. To facilitate exploitation of heterogeneity of architectures, several implementations of a codelet can be supplied by the application developer. Based on auto-tuned performance models, the runtime decides on an appropriate scheduling strategy for their execution.

For the data management part, a high level library was designed that takes care of jobs like transferring data between accelerators and making sure that every accelerator has a consistent view of the data.(Augonnet and Namyst, 2008) Using an MSI caching protocol, StarPU ensures that data does not have to be moved between accelerators and processors explicitly and so-called *filters* help to indicate subsets of data that can be processed separately. Applying filters is done by application programmers, since usually they know best about the structure and dependencies of the data.(Augonnet et al., 2011) The runtimes features can be seen in Table 2.2.

| | Data Distribution | Scheduling on Shared Memory | Scheduling on Distributed Memory | Performance Monitoring | Fault Tolerance | Target Architecture |
|---|---|---|---|---|---|---|
| StarPU | e | **m** | **x** | **off/on** | **x** | **d** |

Table 2.2: StarPU features
**i**...implicit, **e**...explicit, **m**...multiple (incl. ws), **l**...limited, **ws**...work stealing, **tf**...task faults, **pf**...process faults, **sm**...shared memory, **d**...distributed memory, **on**...online use, **off**...offline use (Thoman et al., 2017, p. 9)

### 2.2.3 Legion

The Legion programming system is developed at the Stanford University together with Los Alamos National Laboratory and NVIDIA and was introduced in 2012.(Bauer et al., 2012) The main driver for the development of this project is the aim for better portability between different heterogeneous systems. Legion is described as a "data-centric parallel programming system" that enables developers to not only describe the parallelism, but also the structure of the program data. Thus, Legion uses all of this information to make a decision about the most efficient placement of data and tasks.
The architecture of the Legion programming system actually consists of two runtime systems, a low-level runtime called Realm upon which the higher-level Legion runtime is attached.(Aiken et al., 2014)

The following three abstractions are central in the Legion runtime: *tasks, regions, mapping*. **Tasks** - as in many other parallel programming models too - form the basic unit of parallel execution. In the beginning of program execution a directed acyclic graph is constructed that reflects the dependencies among different tasks. Indicating on what data a task works is done via **regions**.
A **mapper** then assigns tasks to processing elements and regions to appropriate memory (accessible to corresponding tasks). Together with the scheduler, the tasks are then scheduled for execution. Based on the fact that a lot of effort goes into build-up of the dependency graph and the mapping of tasks to processing elements, one would expect computational overhead at runtime leading to worse performance. But to hide this latency, graph-creation happens in advance such that tasks can immediately be scheduled when ready to run.
As a communication means on distributed machines Legion uses GASnet and CUDA for the GPU-accelerated computing on heterogeneous systems. (Bauer, 2014)

Table 2.3 shows the features of Legion based on the taxonomy introduced previously.

| | Data Distribution | Scheduling on Shared Memory | Scheduling on Distributed Memory | Performance Monitoring | Fault Tolerance | Target Architecture |
|---|---|---|---|---|---|---|
| Legion | **i** | **ws** | **ws** | **off/on** | **pf** | **d** |

Table 2.3: Legion features
**i**...implicit, **e**...explicit, **m**...multiple (incl. ws), **l**...limited, **ws**...work stealing, **tf**...task faults, **pf**...process faults, **sm**...shared memory, **d**...distributed memory, **on**...online use, **off**...offline use (Thoman et al., 2017, p. 9)

The features of the Legion runtime are almost identical to the ones by Charm++. The only difference is that for shared memory and distributed memory systems not multiple schedulers exist, but only a work-stealing one.

### 2.2.4 HPX

The HPX runtime is based on the concepts of the paralleX execution model[5]. Since 2007 it is being developed at Luisiana State University.
The developers of HPX have compiled a set of factors that according to them prevent good scaling and thus form the driving factors in the design of their runtime - the *SLOW* factors: **s**tarvation, **l**atencies, **o**verheads and **w**aiting for contention. As countermeasure for these factors, certain design principles were followed.
Instead of avoiding latencies altogether, *latency hiding* was tried to achieve. In order to achieve latency hiding also for short operations, it is desirable to have *lightweight threads* which allow for fast context switching. *Constraint-based synchronization* is used in order to reduce unnecessary global barriers that pose an "eye of the needle" all threads have to be squeezed through. Using a global and uniform address space, HPX makes sure that *dynamic and adaptive locality control* can be achieved. Data transfers are kept at a minimum at different levels like taking advantage of CPU caches or reducing data transfers to the GPU. But HPX also tries to reduce the amount of bytes transferred between nodes, where it can sometimes be advantageous to *transfer work to the data* rather than transferring the data to where work shall be performed. As a means to reduce the synchronization of senders and receivers of data being exchanged, HPX favors message-driven communication over message-passing communication. (Kaiser et al., 2014)

Table 2.4 lists the features of the HPX runtime.

| | Data Distribution | Scheduling on Shared Memory | Scheduling on Distributed Memory | Performance Monitoring | Fault Tolerance | Target Architecture |
|---|---|---|---|---|---|---|
| HPX | i | **ws** | **x** | **off/on** | **x** | **d** |

Table 2.4: HPX features
**i**...implicit, **e**...explicit, **m**...multiple (incl. ws), **l**...limited, **ws**...work stealing, **tf**...task faults, **pf**...process faults, **sm**...shared memory, **d**...distributed memory, **on**...online use, **off**...offline use (Thoman et al., 2017, p. 9)

[5]HPX = High Performance ParalleX

### 2.2.5 OpenMP

OpenMP is an open standard which was first released in 1997.(Dagum and Menon, 1998) It defines a set of compiler directives and runtime library functions that enable application developers to mark regions in their code that can be executed in parallel. In case a program enhanced with OpenMP compiler directives is compiled with a compiler that does not support OpenMP, the directives are simply ignored and one ends up with a sequential version. OpenMP only supports shared memory parallelism and would have to be complemented with a distributed programming model like MPI if distributed memory functionality is required.

OpenMPs execution model follows the fork-join scheme. An OpenMP program has a pool of threads at its disposal, out of which in the beginning only one thread, the so-called *main thread* is executing. Once the program reaches a *parallel section* - which is marked as such by placement of corresponding directives - execution forks and work is distributed among the threads in the thread pool. At the end of a parallel section - again, marked as such via directives - an implicit barrier leads to a join operation after which only the master thread continues executing.

Whereas in the early stages of the OpenMP standard only loops could be parallelized, the scope increased over time and more and more features entered the specification. One of the features added in version 3.0 of the standard is the `task` directive. This directive adds the possibility to express irregular parallelism as opposed to the previously existing directives whose focus lay more on regular parallelism. (Copty et al., 2008)

A description of the current state of the standard can be found in (OpenMP Architecture Review Board, 2018).

Table 2.5 summarizes the features of OpenMP according to chapter 2.1.

| | Data Distribution | Scheduling on Shared Memory | Scheduling on Distributed Memory | Performance Monitoring | Fault Tolerance | Target Architecture |
|---|---|---|---|---|---|---|
| OpenMP | x | **m** | **x** | **off/on** | **x** | **sm** |

Table 2.5: OpenMP features
**i**...implicit, **e**...explicit, **m**...multiple (incl. ws), **l**...limited, **ws**...work stealing, **tf**...task faults, **pf**...process faults, **sm**...shared memory, **d**...distributed memory, **on**...online use, **off**...offline use (Thoman et al., 2017, p. 9)

As OpenMP is shared-memory only, there is no scheduling for distributed memory setups.

### 2.2.6 Open Community Runtime

The Open Community Runtime (OCR) is a project developed within the Department of Energy's XStack project with the aim to develop an asynchronous task-based runtime system for extreme scale parallel systems. With its first release being in 2012 the OCR is - compared to the other runtime systems - a quite young project. Several implementations of the OCR specification exist, among which one is OCR-Vsm[6] developed at the University of Vienna.

In the OCR, programs are composed of so-called *event driven tasks* (EDT), which can be connected via dependencies. All data handling happens through *data blocks*, which are contiguous arrays of bytes whose size does not change once fixed. Together, these OCR objects form a DAG representing the corresponding program.

Table 2.6 shows the features of the Open Community Runtime based on the taxonomy introduced previously.

| | Data Distribution | Scheduling on Shared Memory | Scheduling on Distributed Memory | Performance Monitoring | Fault Tolerance | Target Architecture |
|---|---|---|---|---|---|---|
| OCR | **i** | **m** | **x** | **x** | **tf** | **sm** |

Table 2.6: OCR features
**i**...implicit, **e**...explicit, **m**...multiple (incl. ws), **l**...limited, **ws**...work stealing, **tf**...task faults, **pf**...process faults, **sm**...shared memory, **d**...distributed memory, **on**...online use, **off**...offline use

Several of the features used by the taxonomy are not covered by the OCR standard. Therefore, the specific implementation of the OCR used for the experiments in this thesis - OCR-Vsm - will be described. Concerning the data distribution, the application developer has to specify distribute the data into data blocks, but does not explicitly have to distribute it among cores. Thus, it is still being seen as implicit data distribution. OCR-Vsm uses a work-stealing scheduler and specifically targets shared memory machines. Albeit there is a distributed version developed at the University of Vienna, called OCR-Vdm.

As OCR-Vsm is a fairly young implementation of the Open Community Runtime, at the

[6]available at https://www.univie.ac.at/ocr-vx/

moment neither offline nor online monitoring does exist. Task faults can be detected by the runtime.(Mattson et al., 2016b)

# Chapter 3

# Methodology

In this chapter the applications that were used to perform the experiments shall be described. After that, the coupling mechanism introduced by Dokulil and Benkner (Dokulil and Benkner, 2018) that was used for the co-scheduling of the applications will be explained.

## 3.1 General

Scientific applications pose a high computational load on computers. They run on high-performance computers for good reasons. But these applications are not just running on very powerful machines to speed up execution, they are usually also highly optimized for certain architectures. So, porting such an application to a new architecture takes a considerable amount of development effort.
To judge whether a certain scientific application would benefit from a new architecture, so-called proxy applications have been developed in recent years. They are simplified scientific applications, that behave the same as their full-blown counterparts with respect to their computational load, but only contain a fraction of the code which makes them easier to port to different supercomputer architectures. As these proxy applications mostly use the same algorithms, by running them on new architectures one can also find hot spots that would benefit from new algorithmic approaches.

In this work, such a proxy application called *CoMD* is being used for the experiments and will be described in the following chapter.
The counterpart to the CoMD application in the co-scheduling scenario is a simple ray-tracer application that continuously renders the atoms at their positions in space.
The final section of this chapter describes the OCR coupling mechanism introduced by

(Dokulil and Benkner, 2018) that enables two instances of the runtime to be coupled and co-scheduled. This mechanism is currently only available in the OCR implementation used in this thesis - OCR-Vsm[1].

[1]available at https://www.univie.ac.at/ocr-vx/

## 3.2 Open Community Runtime

The Open Community Runtime (OCR) is the runtime system that is used for the experiments in this document and thus will now be discussed more thoroughly.

The initial release of the OCR was in 2012 at the Supercomputing2012 conference and since then it has been developed further constantly.
The aim of the OCR is not so much to be a production ready runtime, but rather a tool for performing research in the realm of asynchronous many-task runtime systems.
The standard defining the Open Community Runtime is developed under collaboration of various institutions, most notably the Rice University, Intel Corporation and University of Vienna.

An application written using the Open Community Runtimes is represented as a directed acyclic graph (DAG). Here, the nodes of the graph represent tasks and the edges represent dependencies between the tasks. These dependencies define the order in which the tasks can be executed. Once all the dependencies of a task (incoming edges of the node) are satisfied, the task is ready and will at some point be run. It is not guaranteed though that it will be executed immediately. Once the task is running, it will run to completion because no matter what other tasks do, its own dependencies are already fulfilled and it has to finish without blocking.

There are different programming models that can be implemented using the OCR, like data-flow, fork-join and bulk-synchronous programming. This is due to the fact that it has a rather low-level application programming interface and for example does not have an API routine performing a `reduction` operation. The standard therefore also does not call it a primary interface for application level programming. In contrast to, for example, Legion, the concurrency of an application written using the OCR has to be explicitly expressed.

As the OCR standard does only give minimal constraints concerning the hardware, the Open Community Runtime can basically run an almost any hardware it is ported to.

### 3.2.1 OCR Objects

The three fundamental objects used to develop applications that are managed by the OCR are *Event Driven Tasks (EDT), Data Blocks and Events*. These will be described in the following subsections. But before, the concept of slots needs to be discussed, as this is of relevance for the connection of OCR objects.

**Slots** are the term used in the OCR for defining data dependencies and control dependencies between OCR objects. There exist pre- and post-slots. If there is a dependency between two objects A and B with the edge pointing from A towards B, meaning B depends on A, then the post-slot of object A would be connected to the pre-slot of object B.
These slots can be in three different states, which are:

- **Unconnected**, if there is no dependency specified
- **Connected, but unsatisfied**, if a dependency is specified, but the condition under which the dependent object is notified is not yet satisfied
- **Connected and satisfied**, if a dependency is specified and the condition under which the dependent object is notified is satisfied

Out of all the slots an OCR object can have, exactly one of them is a post-slot and zero or more are pre-slots. Using the above example with a dependency between objects A and B, a data dependency can be achieved by returning the data block GUID from the EDT function of object A and thus associating it with its the post-slot. This leads to the data block being available to object B.

Another important entity is the **Global Unique ID (GUID)**. As the name already suggests, this is an identifier for OCR objects that is globally unique. Creation and management of the GUID is handled by the runtime and the application developer does not have to deal with this. GUIDs are the handles used to work with OCR objects. The developer receives a GUID from the API call that creates the OCR object and uses it as input to other API calls.

#### Event Driven Tasks (EDT)

As the Open Community Runtime is a task-based runtime, the basic units of work that are executed in parallel are tasks. In the OCR these are called *Event Driven Tasks*.

EDTs are defined as functions with a certain interface. Their input arguments consist of an array of a variable number of 64-bit parameters and a variable number of dependences on other OCR objects.
The return value of an EDT is a GUID. This has to be either a NULL GUID or a data block GUID. The post-slot of the EDT is then being satisfied by this returned GUID. As mentioned before, post-slots are used to specify dependences between OCR objects.

### Data Blocks

Any data used within an OCR program that needs to be shared among EDTs needs to be put in a *Data Block (DB)*. Like EDTs, they get assigned a GUID that is globally unique. But although the GUID is theoretically accessible to all other OCR objects, EDTs can only access data blocks under two circumstances: the data block is either passed in via a pre-slot or the EDT is the creator of the DB.
In memory, data blocks are contiguous chunks. As it is guaranteed that the memory of no two data blocks is overlapping, one can safely access the contents of them using pointers when knowing the start adress, an offset and the size of it.
But before one can access a data block, it first needs to be acquired. Otherwise the pointer to the start adress is not yet valid. When accessing the data, this can only be done whithin the access rights specified at acquisition of the data block.

The are five possibilities:

- *Read-Write*: This is the default case, i.e. when nothing else is specified. An EDT that has acquired the DB can read and write to it, but not exclusively! All other EDTs can read and write to/from it as well, so data races can occur. This is something the programmer has to take care of.

- *Exclusive-Write*: Here, the same applies as with *Read-Write* mode, but access is exclusive. That means that no two EDTs can acquire a data block and write to it at the same time. Only once the first EDT has released the DB, the second may acquire it.

- *Read-only*: When an EDT is acquiring a data block with read-only mode it claims to only read from it, but technically is not hindered by the runtime to also write to it. However, the behaviour of subsequent reads to that data block by other EDTs is undefined.

- *Constant*: In this mode an EDT can read from the data block and the runtime makes sure that no conflicting write from any other EDT becomes visible.
- *NULL*: This mode can be used when the EDT has no intention of accessing the data block. The GUID however will still be passed to the EDT.

### Events

Events are the means provided by the OCR that can be used to bring order in the execution of tasks. This is done by connecting the post-slot of an event with the pre-slot of other OCR objects and thus creating a dependence between the two. Once the event is satisfied, the objects attached to its post-slot are notified. With this mechanism, synchronization among different tasks can be achieved and a more unstructured parallelism can be expressed than with only using dependences between EDTs.
Like the other OCR objects, they have a single pre-slot and one or more post-slots. The actual number of post-slots depends on the type of event.

There are rules for when events are being triggered. The default trigger rule for events is that their post-slot gets triggered, when any one of their pre-slots is satisfied. However, the *Latch* event behaves slightly different, which will be described below. Additionally, data blocks can be associated with the pre-slot of an event. When the event is satisfied, the GUID of the data block is passed through at the post-slot, enabling the implementation of data-flow algorithms.

The different types of events are:

- *Once event*: This event is destroyed once it is satisfied. Thus, objects having a once event linked to their pre-slot must already be created and linked to the event at the time it is satisfied.
- *Idempotent event*: Only the first attempt to trigger the event has an effect. Any subsequent try is being ignored. This event does not destroy itself, but has to be destroyed explicitly by a call to `ocrEventDestroy()`.
- *Sticky event*: An `ocrEventDestroy()` has to be performed in order to destroy the event. Like the idempotent event, it cannot be triggered multiple times with the difference that any attempt to trigger the event after the first one results in an error.

- *Latch event*: Like the once event, the latch event is destroyed after being triggered. The triggering of this event is somewhat different. It has two pre-slots, an increment and a decrement slot. Its post-slot will only be triggered when an equal number of satisfies for both of the pre-slots are fired.

### 3.2.2 Execution Model

The starting EDT of every OCR program is the so-called `mainEdt()` function. This function has to be provided by the application developper so the OCR can find its entry point. Execution of the program ends, when `ocrShutdown()` or `ocrAbort()` is called.

Figure 3.1 shows the states an EDT goes through during its lifetime. After an EDT is created it is in the *Available* state. The runtime provides its GUID, which is already usable at that point.
Once all the dependences are defined, the EDT becomes *Resolved*. The transition from *Available* to *Resolved* is hard to pinpoint, as any time further dependencies can be added. When all its pre-slots are satisfied, the EDT is ready to be executed and transitions into the *Runnable* state. In order to get into the *Ready* state, all the data blocks associated with the EDT need to be acquired.
Going from *Ready* to *Running* is guaranteed to happen, but it depends on the runtime when exactly this happens.

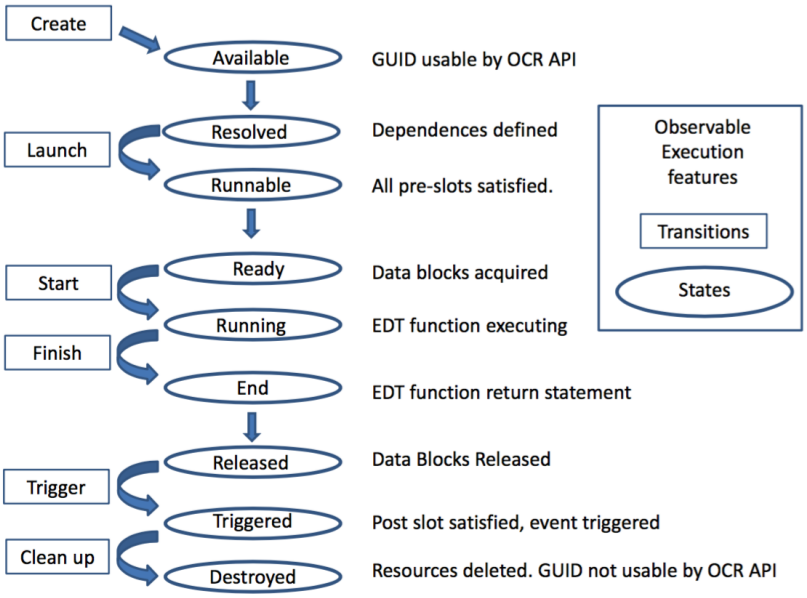



Figure 3.1: EDT execution states (Mattson et al., 2016b, p. 14)

When the task has finished the execution of its work it transitions to the *End* state and once all data blocks are released again into the *Released* state. After releasing the data blocks, other EDTs are able to see changes made to these data blocks[2]. After

[2]In case of RW and RO data blocks, changes to these data blocks could potentially be visible already before.

this, the post-slot of the EDT can be satisfied and the event associated with the EDT triggered, resulting in the output event of the EDT being *Triggered*. Now it is up to the runtime to clean up and delete resources used by the EDT. Afterwards the EDT is *Destroyed*.

The lifetime of OCR objects starts with one of the various create functions, e.g. *ocrDbCreate()*. The GUID returned by this API call is valid immediately and can be used for further actions. The object creation itself can be deferred by the runtime to a later point.
The end of the object lifetime can either manually be summoned by a call to e.g. *ocrDbDestroy()*, or happens automatically.
While data blocks need to be destroyed explicitly, the *once* and *latch* event are destroyed automatically after being satisfied.

### 3.2.3 Memory Model

In the Open Community Runtime the memory model has the notion of *synchronized-with, sequenced-before* and *happens-before* relationships between operations.
Two EDTs have a *synchronized-with* relationship, when there are constraints that request them to be executed in a certain order. The OCR means to achieve this behavior are events.
The *sequenced-before* relationship on the other hand is concerned with the order of two operations within an EDT. This order is defined by the programming language. The combination of these two relations results in the *happens-before* relation and makes it possible for the runtime to make certain guarantees in its memory model.
In essence, the OCR guarantees that in a scenario with two EDTs having a *happens-before* relationship, if the release happens-before the acquire, the acquiring task sees all the changes made to the data block by the other task.

## 3.3 CoMD

CoMD is a proxy application that shall represent the computational structure of classical molecular dynamics simulations. *ddcMD* and *Scalable Short-range Molecular Dynamics (SPaSM)* are the two codes that CoMD is based on. The OCR version of CoMD comes with the OCR implementation available at https://xstack.exascale-tech.com/git/public/ocr.git and only slight modifications had to be made to be able to run the application with OCR-Vsm.

Molecular dynamics simulations numerically solve Newton's equations of motion for a large ensemble of atoms. So the movement of the atoms is calculated, based on the forces acting upon every single atom. These forces are defined as potentials that depend on the distance between the atoms. From a certain distance upwards, the interaction can be neglected. This distance is the so-called cut-off distance. The problem here is that in order to know which atoms fall within this cut-off radius, one has to go through all atoms and calculate the distance to the atom of interest. This results in a computational complexity of $O(N^2)$.
Rectangular spatial decomposition is not only used for parallelization, but also comes in handy as a remedy for the computational complexity problem (see Figure 3.2).

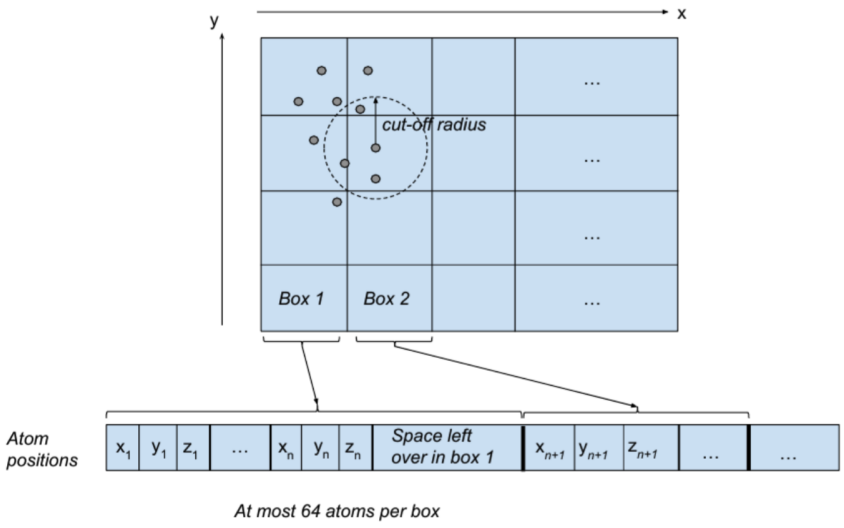



Figure 3.2: CoMD spatial decomposition Bjørnseth et al. (2016)

If the space is divided into boxes that only contain a small subset of the overall atoms, but that are larger than the cut-off radius in size, the search for neighboring atoms can be restricted to the cell the considered atom lies in and the surrounding 26 cells, leading to a $O(N)$ complexity.

Figure 3.3 shows the main timestep loop of CoMD, which consists of a `topEDT` – `botEDT` pair. The `topEDT` starts an EDT for the computation of the force, which in turn starts position computation whose continuation is the velocity computation. Once velocity computation is finished, the `botEDT` either spawns a new `topEDT` or ends the calculation if the convergence criterion is reached.

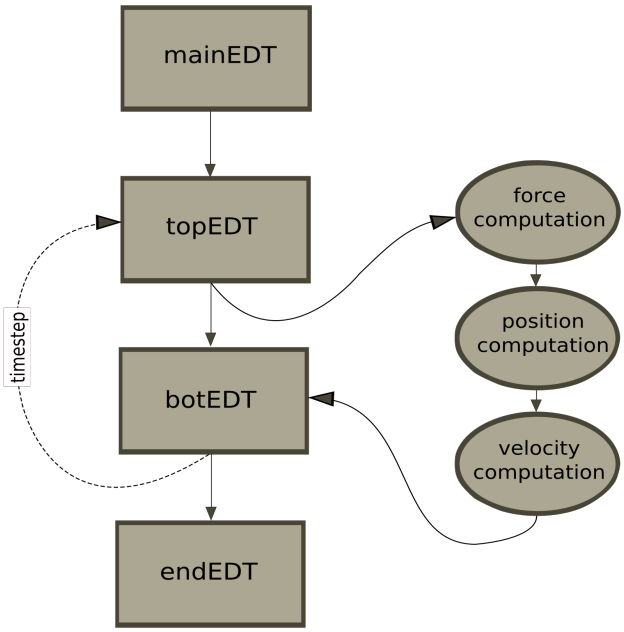



Figure 3.3: CoMD timestep loop

The structure of the computational phases for force, position and velocity is the same and can be seen in Figure 3.4. For each cell a `forkEDT` starts an `updateEDT` which performs the necessary calculations and whose continuation EDTs are set to a single `reduceEDT` that updates the potential of the system and starts the `botEDT`.

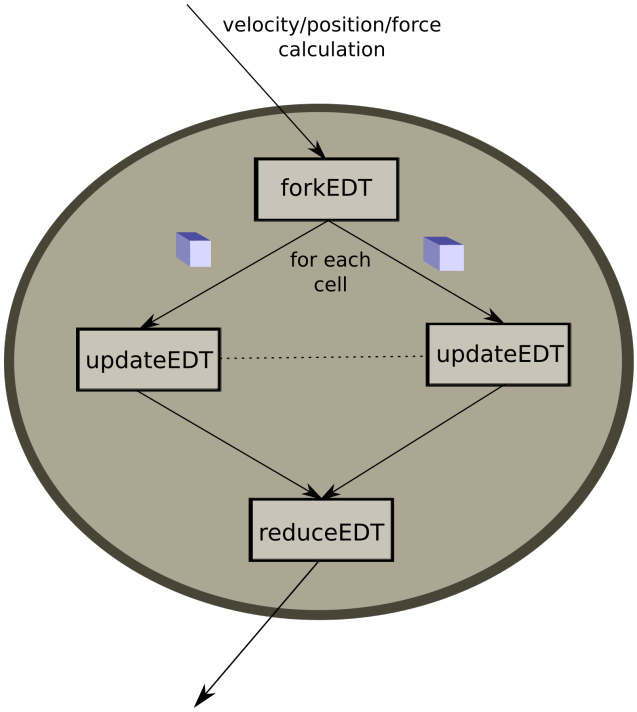



Figure 3.4: CoMD fork/join phase

The data of the individual boxes/cells are stored in separate OCR data blocks, which are accessible via a list that groups them together and thus represents the whole computational domain. (Borkar, 2015, p. 18 ff.)

Every tenth iteration CoMD outputs a single text file in the `.XYZ` file format[3] containing the coordinates of all the atoms.

[3] https://en.wikipedia.org/wiki/XYZ_file_format

## 3.4 Raytracer

For the raytracer OCR application, a simple C++ raytracer implementation was taken from https://www.scratchapixel.com/code.php?id=10&origin=/lessons/3d-basic-rendering/minimal-ray-tracer-rendering-simple-shapes. This code was used as a basis and was subsequently ported to the Open Community Runtime.[4]
Figure 3.5 shows the domain decomposition of the raytracer application. Again, rectangular spatial decomposition is used, but in this case it is only a two-dimensional decomposition.

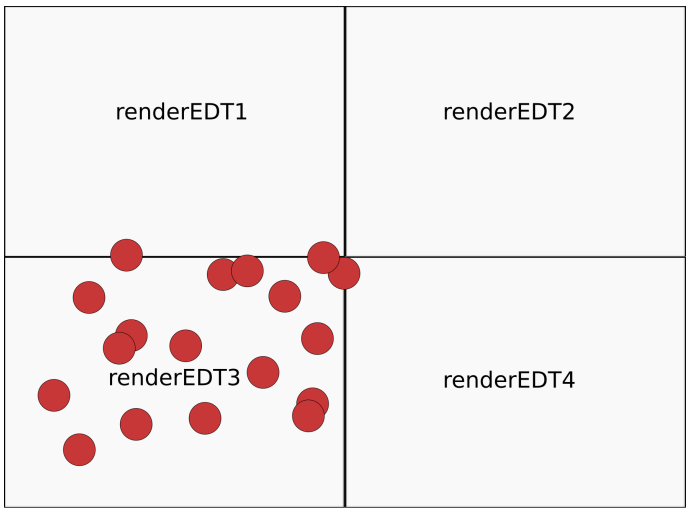



Figure 3.5: Raytracer exemplary 2x2 domain decomposition

At the beginning of each iteration, the presence of the corresponding `.XYZ` file mentioned in the previous chapter is checked. If the file is there, it is opened and the atom positions are read and filled into a single data block.
Additionally, the `mainEDT` creates EDTs for rendering and writing data as image files to disk. The program flow can be seen in Figure 3.6. The `mainEDT` creates one `writeImageEDT` EDT and as many `renderEDT` EDTs as there are subdomains and sets up the dependencies between them. In addition to the data block containing the atom positions for every subdomain, a `framebufferDb` data block is created and a dependency to the corresponding `renderEDT` is set up.

[4]For the basic mode of operation of a raytracer, the reader shall be forwarded to https://en.wikipedia.org/wiki/Ray_tracing_(graphics).

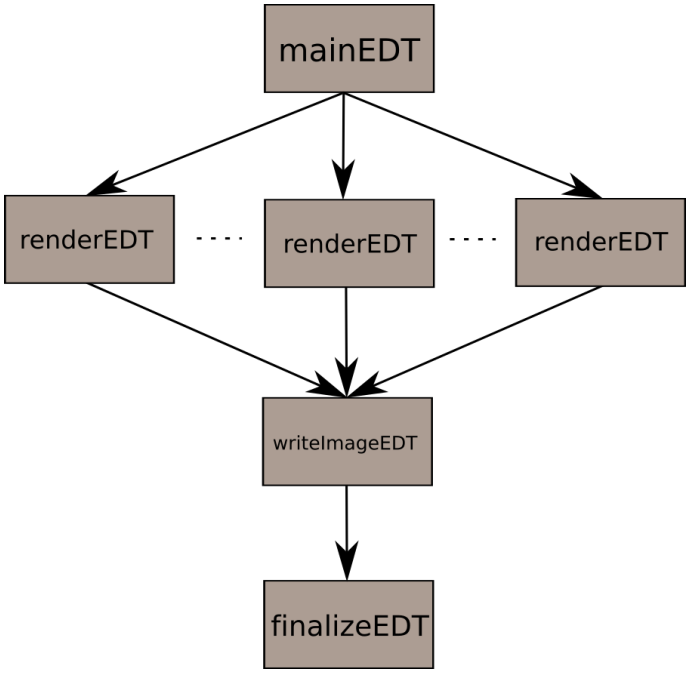



Figure 3.6: Raytracer flowchart

The `renderEDT` renders all the pixels in its subdomain and when finished, releases the `framebufferDb` data block. Following, when all `renderEDTs` are done with their work, the `writeImageEDT` is started and stitches all the framebuffers of the subdomains together and writes the resulting rendered image out as `.ppm` file[5].
Figure 3.7 shows an example of such a rendered image.

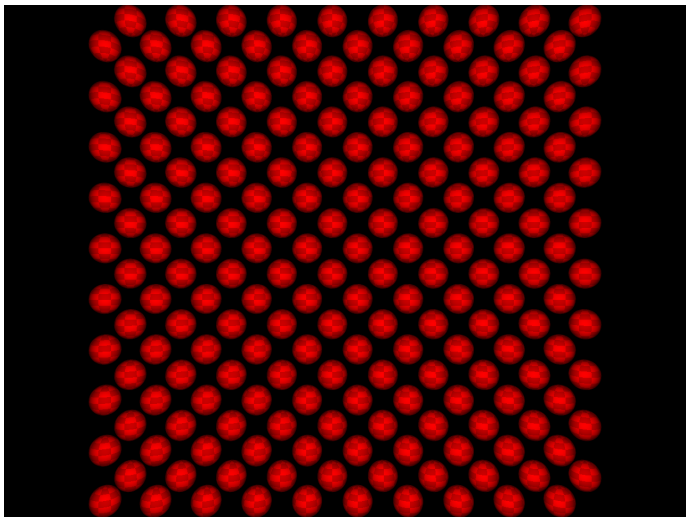


Figure 3.7: Raytracer example rendering

The raytracer application is embarrassingly parallel. Once the data to be rendered is

[5] https://en.wikipedia.org/wiki/Netpbm_format

distributed there is no communication or exchange of data performed between the event driven tasks during the rendering stage.

## 3.5 Coupling of Applications

In (Dokulil and Benkner, 2018) an approach to couple applications using the Open Community Runtime is presented. This makes it possible to run two separate instances of applications using the OCR to run on the same machine and be scheduled by a central agent.
As this mechanism is being used for the execution of the experiments conducted for this thesis, a closer look will be taken at it in the following.
The OCR task scheduler decides where and when tasks are executed. In (Dokulil and Benkner, 2018) this task scheduler was extended such that the resources assigned to each running instance - and thus process - of the OCR can be changed dynamically at runtime.
Initially, a thread pool of the same number of threads as there are logical cores is assigned to each process. In order to make the dynamic adjustment of the amount of worker threads more efficient, the actual number of threads is not changed. Instead, threads are blocked, telling the operating system not to schedule these threads on any core.

After every executed task the threads check whether they should block or can continue executing a new task. Two atomic counters are used to indicate the desired and the currently used number of threads. When a thread notices that it should block, a condition variable is used to block it and the counter for the number of currently used threads is decremented. The range of active threads goes from 0 to 24.

In order to introduce co-scheduling, a central agent was created, to which running OCR instances connect using the ZeroMQ library (ØMQ).
Additionally to these worker threads, there are management threads. One of these management threads publishes current performance data to the agent and waits for commands to adjust the desired thread count. Together with CPU usage of the applications received via operating system services, the agent can follow different strategies to co-schedule the applications.
An overview of the architecture of this approach can be seen in Figure 3.8.

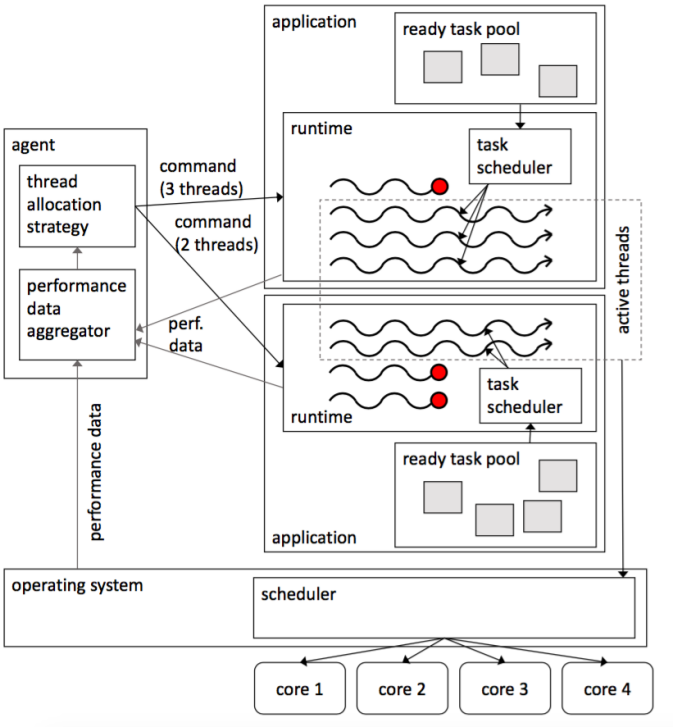



Figure 3.8: OCR co-scheduling architecture (Dokulil and Benkner, 2018, p. 4)

The extension to the OCR API to support this new coupling approach for applications is rather small. Applications need to be extended by only a single function: `ocrProgressReport(u64 progress)`. Pseudocode of a typical simulation/analysis main loop extended with the before-mentioned extension to the OCR API can be seen in Algorithm 1. With this function they provide the agent with their current progress, such that the agent in turn can incorporate this information into its scheduling decisions.

**for** *iteration = 0; iteration < max_iteration; iteration++* **do**
    *perform simulation/analysis tasks*;
    `ocrProgressReport(iteration);`
**end**

**Algorithm 1:** Pseudocode of simulation/analysis main loop extended with `ocrProgressReport(u64 progress)`

# Chapter 4

# Experiments/Results

The proxy application CoMD and the raytracer application introduced in the previous chapter were being used to perform experiments with the coupling approach introduced by (Dokulil and Benkner, 2018).
They form a producer-consumer scenario where the molecular dynamics simulation is acting as the producer and the raytracing application as the consumer.

The machine the experiments were performed on is a Linux server possessing two Intel Xeon X5680 CPUs. Each of these has 6 cores and 12 MB of cache per CPU. The total amount of memory is 24 GB.
As the OCR co-scheduling mechanism uses as many threads as there are logical cores, the maximum number of threads available is 24.

Three different scenarios were looked at:

- the classic, non in situ workflow, where producer and consumer run sequentially
- producer and consumer running concurrently, but not co-scheduled, i.e. with static thread assignment
- and the co-scheduled variant with dynamic thread adjustment

Using the terms to describe in situ processing scenarios introduced in chapter 2.1, the experimental setup can be described as follows. At the end of each iteration, API calls are made to inform the central agent of the current progress of the application. Although this is not exactly a call to an in situ framework, the simulation application is *aware of the analysis application*. As both applications are running on the same machine and

potentially on the same physical core, the *proximity* is *close*. Due to the fact that exchange of data is done via files, the *access* is *indirect*. The resources are divided among the two applications and thus the *division of execution* is *by space*. At the beginning of the execution it is already fixed that the results produced by the analysis application are rendered images which means the *operation controls* axis is *automatic* and *non-adaptive*. Finally, the *output type* is *derived*, the original data is taken and from it rendered images are derived.

In the following, the results of the experiments are presented.

## 4.1 Sequential execution with static resource assignment

In this scenario first the producer application(CoMD) was started and immediately after it had finished, the consumer application was kicked off.
The number of threads each application could use was statically set to 24.

Figure 4.1 shows the computational load genereated by the two applications. One can see, that the load produced by the CoMD application is a lot lower than the load produced by the raytracer.

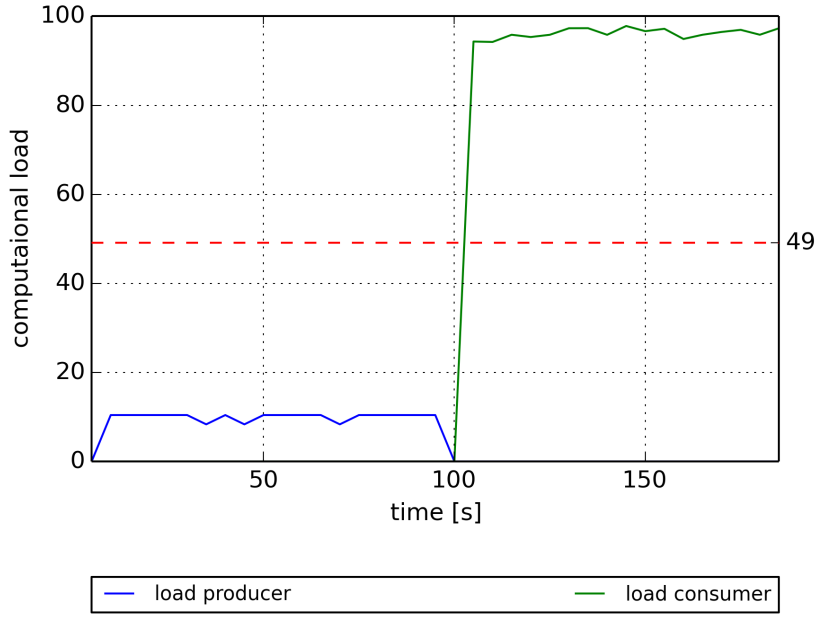



Figure 4.1: Computational load of sequentially executed applications with statically assigned threads.

The raytracer makes use of almost 100 percent of the available computing power, while the simulation only uses about 10 percent. The average use of the computational resources for the duration of the application execution is 49 percent. It takes the two applications 186.75 seconds to finish.

## 4.2 Concurrent execution with static resource assignment

Here, producer and consumer are started at the same time, but like in the sequential scenario the number of assigned threads is static. Instead of assigning 24 threads to each application, only 12 threads were assigned, due to the applications running at the same time and having to share the logical cores. Because the amount of parallelism that can be exploited by an application is not necessarily known beforehand, resources were divided equally.

The computational load generated in the concurrently running case can be seen in Figure 4.2.

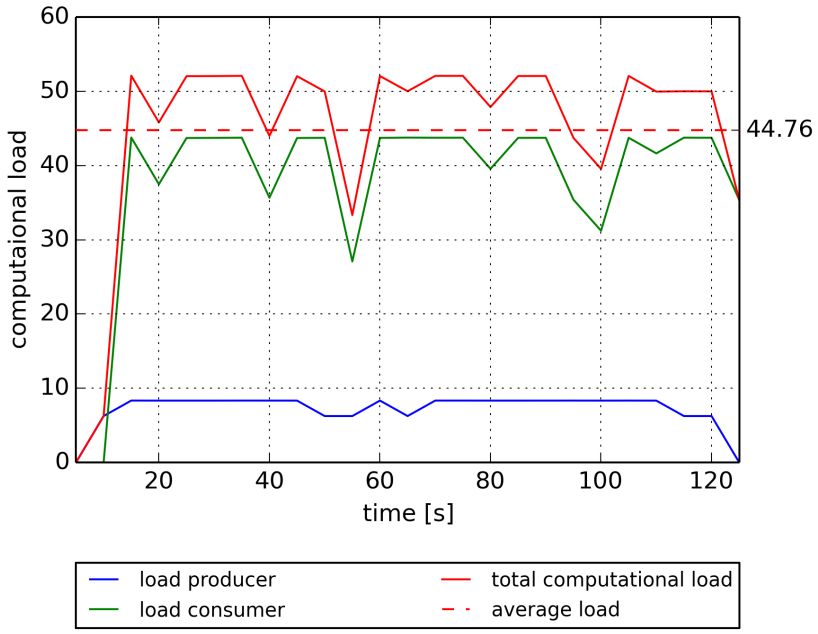



Figure 4.2: Computational load of concurrently executed applications with statically assigned threads.

One can see, that the main contribution to the load is coming from the consumer application. This behaviour of the applications, i.e. the consumer showing a lot higher computational load, could already be seen in the sequential scenario.
The average computational load over the time of execution of the applications reaches around 44 percent.
The progress made by producer and consumer application can be seen in Figure 4.3.

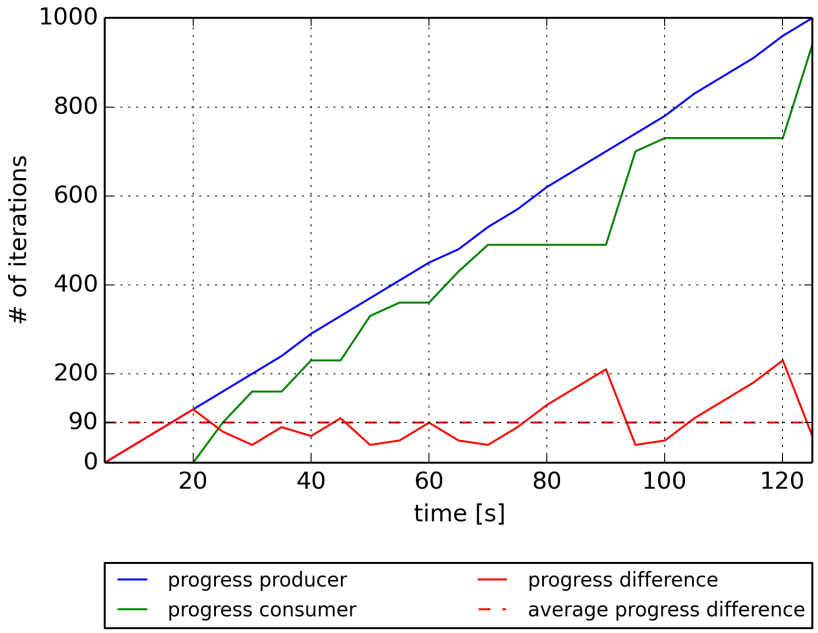
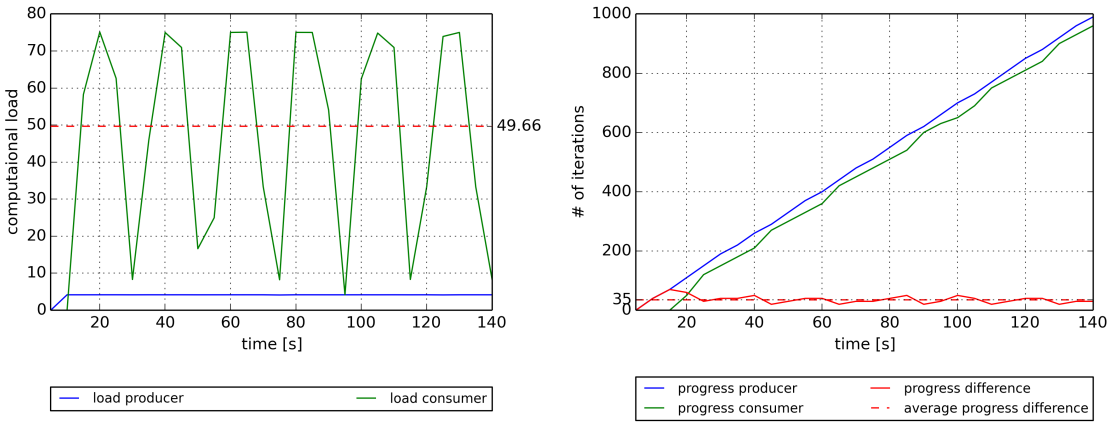



Figure 4.3: Progress of concurrently executed applications with statically assigned threads.

The progress of the producer increases steadily and in the beginning the progress difference is rather small, but in the second half of the excution period at times it increases to up to a difference of 200 iterations. Despite these spikes in the progress difference, the average difference lies at 90 iterations.

The results from the experiments seen so far suggest that instead of equally dividing the available threads among the applications, assigning more threads to the consumer application and less to the producer application could result in a better usage of the available computational resources.
Thus, another experiment was performed with assigning 4 threads to the producer application and 20 threads to the consumer application. The results of this run can be seen in Figure 4.4.




(a) Computational load (b) Progress

Figure 4.4: Results of statically assigning 4 threads to the producer and 20 threads to the consumer.

In Figure 4.4a one can see that the average load indeed has increased to 49.66 %. The progress difference seen in Figure 4.4b has a small peak in the beginning that goes up to 70 iterations, but remains below 50 until the applications finish and has an average value of 35.

The running time though could not be reduced and instead has increased to ∼144 seconds. As the consumer application is embarrassingly parallel, it can make use of all of the 20 threads it has at its disposal and the simulation results are being consumed faster than the producer can generate them leading to starvation of the consumer.

## 4.3 Concurrent execution with dynamic resource assignment

The dynamic and concurrent experiment starts producer and consumer application at the same time. The number of assigned threads is changed dynamically based on the deviation from the current progress difference to a predefined target value and can range from 0 to 24 per application. This means that the runtime does not try to get the progress difference down to zero, but rather to a predefined, tolerated difference, that is called *target value* here.

The scheduling strategy employed to dynamically assign threads to the applications uses a fairly simple heuristic: If the current progress difference is above the target value, the thread count of the consumer is increased, if it is below the target value, the thread count of the producer is increased. In case the progress difference coincides with the target, no action is taken and the thread assignment stays as it is. Additionally, the larger the deviation from the target progress difference, the larger the increase in threads is. The minimum increment/decrement of threads is 2 threads, but with larger target deviations, the increment increases with integer multiples of 2.
As scheduling target, a progress difference of 80 iterations was chosen.

The computational load plot seen in Figure 4.5 shows the same behaviour already seen in the sequential and the concurrent, static scenarios. The load generated by the producer is rather low and lies below 10 percent, while the load generated by the consumer is significantly higher with spikes of up o around 60 percent.

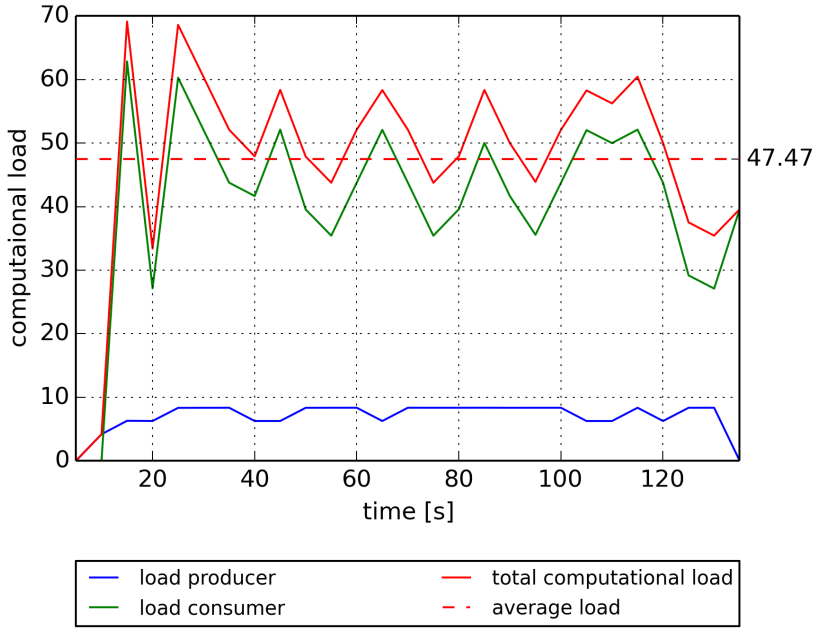



Figure 4.5: Computational load of concurrently executed applications with dynamically assigned threads.

The almost 100 percent computational load of the consumer application seen in the sequential scenario can not be reached, as the application does not have the full amount of threads at its disposal. The dynamic approach reaches an average computational load of ∼47 percent. With a target progress difference of 80 iterations, the progress plot in Figure 4.6 shows that with the dynamic co-scheduling mechanism the average progress difference can be kept within this bound.

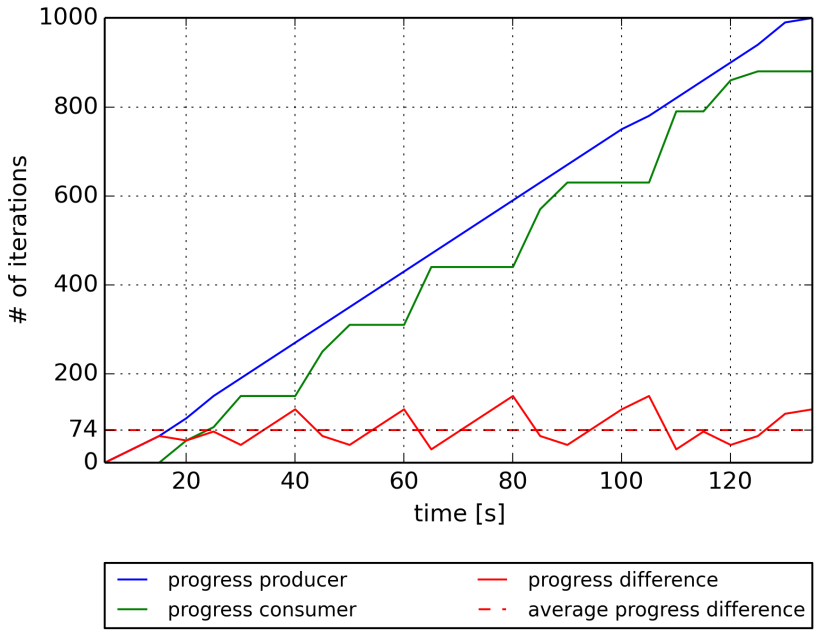



Figure 4.6: Progress of concurrently executed applications with dynamically assigned threads.

There are small spikes in the the progress difference, but those reach a level of at most 140 iterations (this can better be seen in the bottom plot of Figure 4.7). The average progress difference lies at 74 iterations. The upper plot in Figure 4.7 shows the number of threads assigned to producer and consumer application.

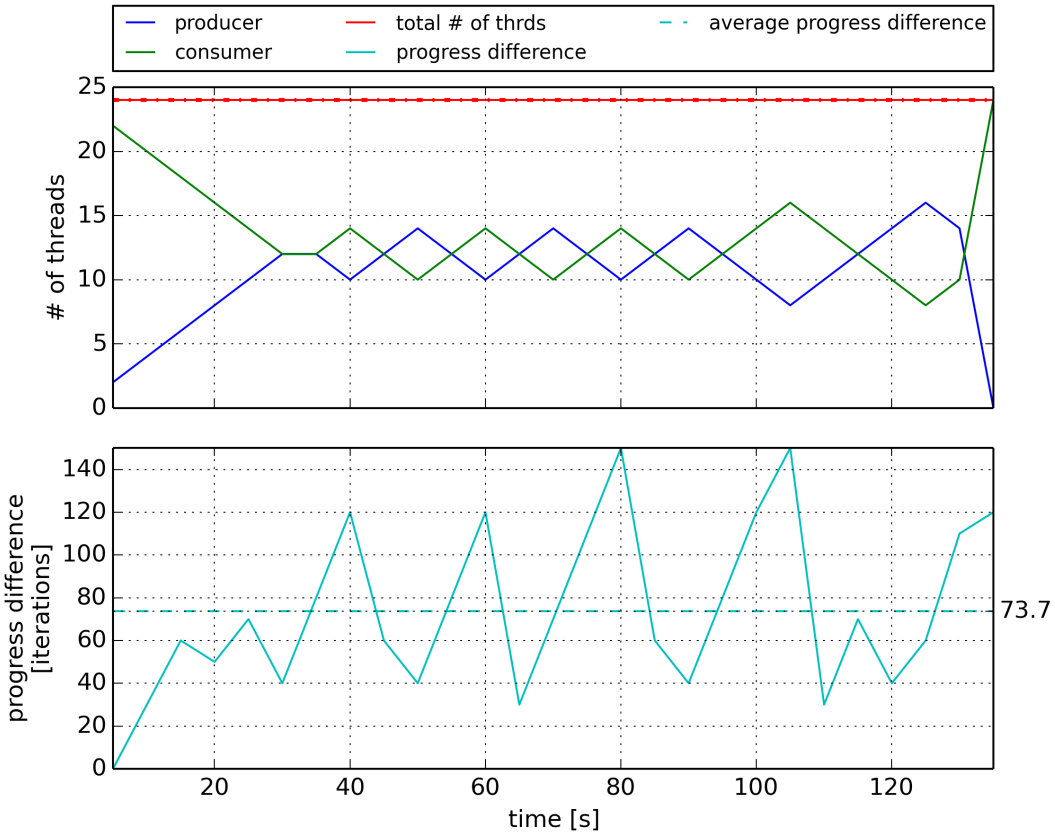



Figure 4.7: Thread assignment of concurrently, co-scheduled applications.

One can see that the scheduling algorithm reacts to deviations from the target progress difference with an increase in threads assigned to the consumer and as soon as the progress difference goes below the target value, reduces the threads again.

In Figure 4.8 one can see the different running times for the above mentioned experiments.

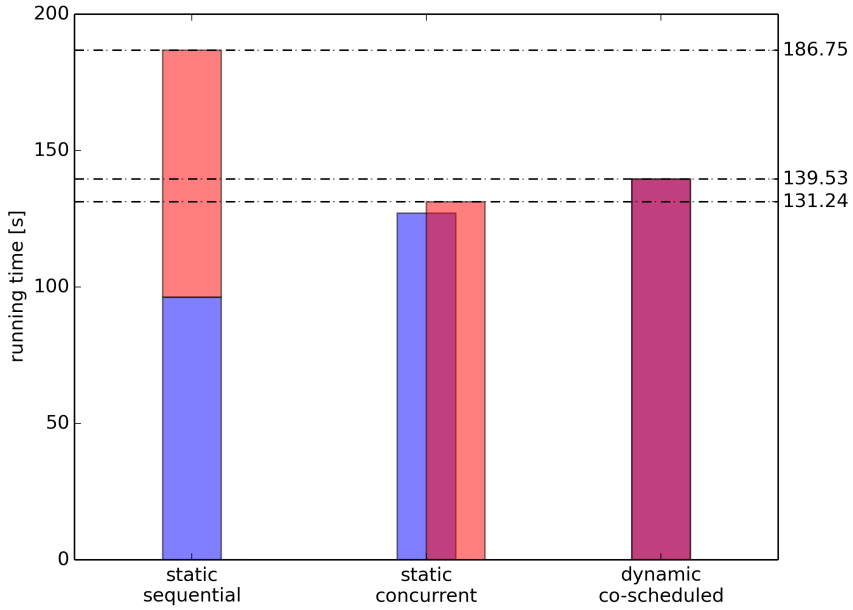



Figure 4.8: Running time comparison between sequential execution with static threads assignment, concurrent execution with static thread assignment and concurrent execution with co-scheduled thread assignment.

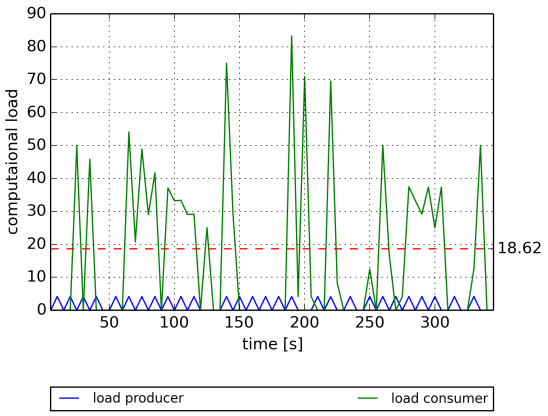
It can be seen that both the concurrent execution with statically assigned threads and the concurrent execution with dynamically assigned threads perform better than the sequential execution. The scenario using dynamically adjusted threads though does not perform best and has an execution time of a few seconds more than the concurrent scenario with static thread assignment. Table 4.1 gives the speed-ups of the two concurrent approaches.

| | speed-up |
|---|---|
| static | 1.42 |
| dynamic | 1.34 |

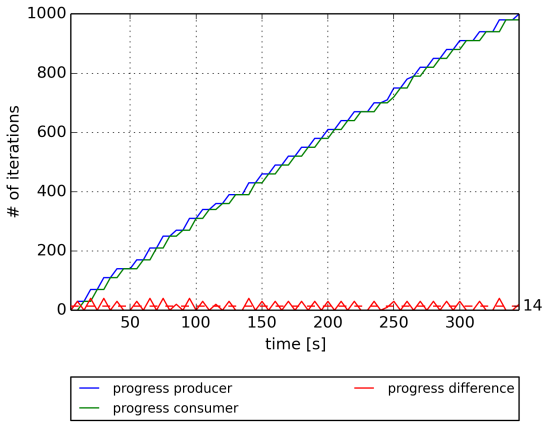
Table 4.1: Speed-up of concurrent execution with static thread assignment and concurrent execution with dynamic thread assignment.

While the static approach achieves a speed-up of 1.42, the dynamic approach reaches only a slightly smaller speed-up of 1.34. This could be explained with the additional scheduling overhead that has to be performed in the dynamic approach.

To put the scheduling heuristic to test and see whether there is the potential for a further decrease in execution time, additional progress differences were set as target. The following plots in Figures 4.9 and 4.10 show the results. In the first experiment, a target value of 10 was used and in the second experiment a target value of 40.




(a) Computational load




(b) Progress

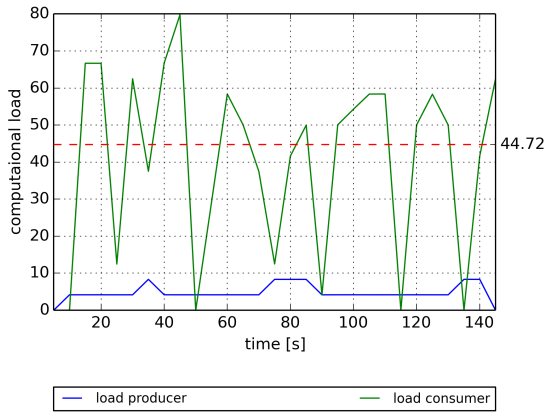

(c) Thread assignment

Figure 4.9: Dynamic scheduling results for a scheduling target of 10 iterations progress difference

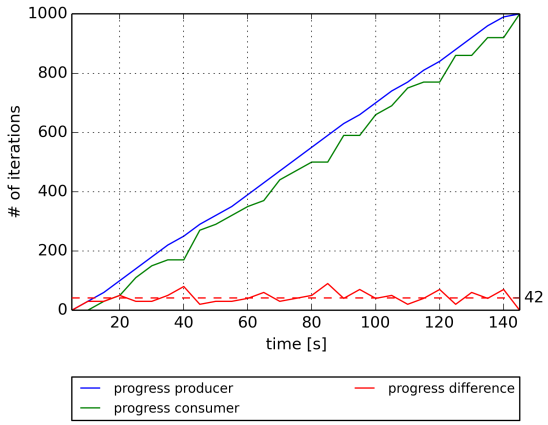
The scheduling algorithm was very well capable of holding close to the target value of 10 iterations progress difference and the average deviation is 14 iterations. But one can also see that the average computational load over the course of the experiment is rather low and lies at ∼18 percent. Although the consumer is constantly assigned at least 20

threads, work generated by the producer only allows it to work for a short period of time, resulting in the spikes that reach up to ∼80 percent computational load, followed by a starvation period.
Although progress is tracked very nicely, execution time suffers under this scheduling target and more than doubles from 139 seconds in the previous experiment to 347 seconds.

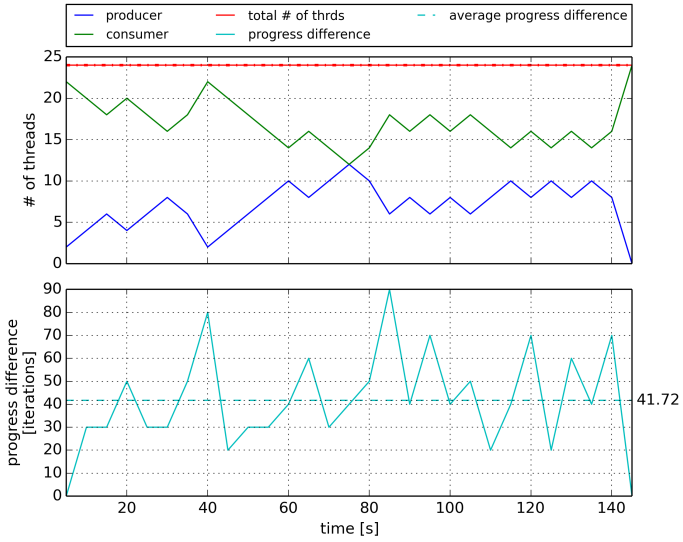
Setting the target to 40 iterations gives better results with respect to execution time. Here, both applications are finished after 145 seconds and thus only ∼5% slower than with the 80 iteration scheduling target.




(a) Computational load




(b) Progress




(c) Thread assignment

Figure 4.10: Dynamic scheduling results for a scheduling target of 40 iterations progress difference

Again, the target progress difference of 40 iterations is followed well and the average deviation is 41.72 iterations. One can see a little more dynamic behavior in the thread assignment which spans from 12 to 24 for the consumer and 0 to 12 for the producer. In Figure 4.10c one can also see the progressively increasing steps of thread increase as the target deviation grows. At the first spike of the progress difference (at about 40 seconds) the amount of threads assigned to the consumer does increase from 18 to 22 to counteract the target deviation.

## 4.4 Discussion of results

The experiments discussed previously showed that a significant performance increase can be achieved by progressing from the traditional workflow with sequential execution of a data-producing and a data-consuming application towards a concurrent in situ approach. Different scenarios with concurrently executing applications and statically assigned computational resources were run. First, resources were evenly divided among the two applications and a speed-up of 1.42 with respect to the sequential scenario was achieved. In another experiment, the results from previous experiment were taken into account, where it could be seen that the producer application is not able to make use of all its assigned resources. Therefore, the producer was assigned 4 and the consumer 20 of the available 24 threads. The result being that the total running time even increased, due to the fact, that the consumer now had so many computational resources that it consumed results faster than the producer could generate them and thus suffered from starvation.

Running an experiment using a co-scheduling approach and dynamically assigning computationl resources to the applications could not further improve the running time. But with a running time of 139.53 seconds compared to the 131.24 seconds of the experiment with concurrently running applications and static thread assignment, the time penalty amounts to only $\sim$6%. This increase in running time can be attributed to the additional effort that is required for the scheduling and assignment of the computational resources. Experiments with 3 different target values have also shown that the heuristic scheduling algorithm is very well capable of keeping the target progress difference close to different values.
Although the co-scheduling approach is slightly slower, the ability of the scheduling mechanism to track arbitrary target progress differences should not be underestimated.

Different scenarios could very well benefit from the dynamic scheduling. Computational steering, as an example of in situ analysis, would benefit from a smaller progress deviation between a producer and a consumer application. Counteractions could be made earlier to prevent simulations progressing towards undesirable directions. Additionally, a scenario where the load of one of the applications varies strongly from one iteration to the next would benefit from this scheduling capability.

# Chapter 5

# Conclusion

In this thesis two Open Community Runtime applications were used to perform experiments that investigate the potentials of a coupling mechanism that allows for co-scheduling of two separately running instances of the runtime in order to stay within certain pre-defined progress difference bounds. An existing OCR proxy application that performs a molecular dynamics simulation and a raytracing application whose parallelization and porting to the Open Community Runtime is one of the outcomes of this work. These two applications have a producer - consumer relationship with the molecular dynamics application being the producer generating output files containing atomic positions and the raytracing application being the consumer reading these files and rendering images therefrom.

To show the effect of co-scheduled execution of the above-mentioned applications, experiments with static as well as dynamic assignment of computational resources wer performed. Additionally, to show the speed-up of executing the consumer application concurrently to the producer application, sequential and concurrent experiments were run.

The performed experiments showed that a significant performance increase can be achieved when performing the analysis of the results that are generated by the producer application in situ. A comparison of static versus a dynamic assignment of computational resources between the two applications led to the conclusion that when running the applications concurrently, the dynamic approach could not outperform the static approach with respect to the running time, but gives a lot more control over the progression of the single applications. It could also be shown that a simple heuristic scheduling algorithm suffices to keep certain pre-defined progress difference bounds.

Good knowledge about the nature of the executed applications or experience gained from running the applications can enable a static assignment of computational resources to lead to good results with respect to running time or progress difference. But gaining this experience may be costly in real-world scenarios, where simulation runs take longer than a few minutes. Also, for certain applications may not be possible to predetermine their computational behavior, because it changes from one execution to the next.
The dynamic approach used in this work has the advantage of being able to also compensate for irregular workloads or other applications running on the same machine competing for computational resources. The possibility to react to varying workloads was shown in (Dokulil and Benkner, 2018). This makes the dynamic approach far more flexible.

Only a simple heuristic scheduling algorithm was used for the performed experiments. Although having shown to be a good enough solution in this case, this may not be true anymore for a workload that shows more irregularity. It may very well be that for different producer and consumer applications a more sophisticated algorithm would be needed, which could be a topic for further investigation.

## 5.1 Possible future improvements

At the moment, the exchange of data between producer and consumer happens via files. each iteration the producer generates a file containing the current spatial coordinates of the atoms. The consumer on the other hand constantly checks, whether a new file is present that it can process. That means that during the execution of the producer and consumer a lot of file system interactions take place.

Ideally, the exchange of data would be handled by the runtime with some form of inter-process communication mechanism. A possible advantage of inter-process communication mechanisms could be that no reads and writes to/from the file system need to be done. All data resides in memory and is read directly from there. If producer and consumer thread run on the same core, it could also be that data is still cached, which would speed up data transfer even more.

There exist several mechanisms to exchange data/messages between processes, each with its pros and cons. In general one distinguishes between message-passing and shared memory mechanisms. In shared-memory mechanisms the communicating processes - as the name already suggests - share a certain region in memory to exchange data or messages. Message-passing mechanisms on the other hand send messages back and forth. An advantage of shared-memory over message-passing is that only once a system call has to be made for the creation of the shared memory region. Message-passing on the other hand is often implemented using system calls, this can lead to worse performance. However, research on IPC mechanisms on multi-core systems shows that shared-memory performance is worse on these systems than message-passing. The reason for this being the additional efforts needed to maintain cache coherence between cores. (Silberschatz et al., 2008, p. 122 ff.)

A performance comparison of different interprocess communication mechanisms was conducted by (Immich et al., 2003). Experiments with five IPC mechanisms showed that the highest throughput was achieved using pipes, followed by FIFO (named pipes), System V messages, sockets and shared-memory. Above a certain message size though, sockets perform better than System V messages. But the better throughput of sockets compared to shared memory can not be attributed to additional effort for maintaining cache coherence, since the system used for the tests was a single core machine, but rather to the fact that shared memory interprocess communication requires additional

actions to synchronize access to the shared memory region. These actions that are controlled by the kernel lead to more frequent blocking of the process.

Of course, scalability across computing nodes does also have to be taken into account and a closer look would have to be taken on restrictions on the amount of data that can be buffered by the above-mentioned IPC mechanisms.
There are also other performance improvements that could be made, like optimizing the way data is stored in the file-based approach. But as also a sequential execution of the producer and the consumer application would run faster using such an approach, this is not of much interest here, as the speed-up would more or less stay the same because sequential as well as concurrent execution would benefit from this improvement. e

# Bibliography


Aiken, A., Bauer, M., and Treichler, S. (2014). Realm: An Event-based Low-level Runtime for Distributed Memory Architectures. In *2014 23rd International Conference on Parallel Architecture and Compilation Techniques (PACT)*, pages 263–275. IEEE.

Augonnet, C. and Namyst, R. (2008). A unified Runtime System for heterogeneous Multi-Core Architectures. In *European Conference on Parallel Processing*, pages 174–183. Springer.

Augonnet, C., Thibault, S., Namyst, R., and Wacrenier, P.-A. (2011). StarPU: A Unified Platform for Task Scheduling on Heterogeneous Multicore Architectures. *Concurr. Comput. : Pract. Exper.*, 23(2):187–198.

Ayachit, U., Bauer, A., Geveci, B., O'Leary, P., Moreland, K., Fabian, N., and Mauldin, J. (2015). ParaView Catalyst: Enabling In Situ Data Analysis and Visualization. In *Proceedings of the First Workshop on In Situ Infrastructures for Enabling Extreme-Scale Analysis and Visualization*, ISAV2015, pages 25–29, New York, NY, USA. ACM.

Bauer, A. C., Abbasi, H., Ahrens, J., Childs, H., Geveci, B., Klasky, S., Moreland, K., O'Leary, P., Vishwanath, V., Whitlock, B., et al. (2016). In Situ Methods, Infrastructures, and Applications on High Performance Computing Platforms. In *Computer Graphics Forum*, volume 35, pages 577–597. Wiley Online Library.

Bauer, A. C., Geveci, B., and Schroeder, W. (2015). *The Catalyst User's Guide v2.0 ParaView 4.3.1*. Kitware Inc.

Bauer, M., Treichler, S., Slaughter, E., and Aiken, A. (2012). Legion: Expressing locality and independence with logical regions. In *SC'12: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, pages 1–11. IEEE.

Bauer, M. E. (2014). *Legion: Programming Distributed Heterogeneous Architectures with Logical Regions*. PhD thesis, Stanford University.

Bjørnseth, B. A., Meyer, J. C., and Natvig, L. (2016). Study of Xeon Phi Performance of a Molecular Dynamics Proxy Application. Technical report.

Borkar, S. (2015). Traleika Glacier X-Stack - Final Scientific/Technical Report. Technical report.

Copty, N., Unnikrishnan, P., Ayguadé, E., Teruel, X., Massaioli, F., Zhang, G., Hoeflinger, J., Lin, Y., and Duran, A. (2008). The Design of OpenMP Tasks. *IEEE Transactions on Parallel and Distributed Systems*, 20:404–418.

Dagum, L. and Menon, R. (1998). OpenMP: An Industry-standard API for Shared-Memory Programming. *Computing in Science & Engineering*, (1):46–55.

Diaz, J., Munoz-Caro, C., and Nino, A. (2012). A Survey of Parallel Programming Models and Tools in the Multi- and Many-Core Era. *IEEE Trans. Parallel Distrib. Syst.*, 23(8):1369–1386.

Dokulil, J. and Benkner, S. (2018). Adaptive Scheduling of Collocated Applications using a Task-based Runtime System.

Dokulil, J., Sandrieser, M., and Benkner, S. (2015). OCR-Vx - An Alternative Implementation of the Open Community Runtime. In *International Workshop on Runtime Systems for Extreme Scale Programming Models and Architectures, in conjunction with SC15. Austin, Texas, November 2015*.

Hennessy, J. L. and Patterson, D. A. (2011). *Computer Architecture, Fifth Edition: A Quantitative Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 5th edition.

Immich, P. K., Bhagavatula, R. S., and Pendse, R. (2003). Performance Analysis of Five Interprocess Communication Mechanisms Across UNIX Operating Systems. *J. Syst. Softw.*, 68(1):27–43.

InSituTermProj (2016). The In Situ Terminology Project. http://ix.cs.uoregon.edu/~hank/insituterminology/. [Online; accessed 20-March-2018].

Kaiser, H., Heller, T., Adelstein-Lelbach, B., Serio, A., and Fey, D. (2014). HPX: A Task Based Programming Model in a Global Address Space. In *Proceedings of the 8th International Conference on Partitioned Global Address Space Programming Models*, PGAS '14, pages 6:1–6:11, New York, NY, USA. ACM.

Kale, L. V. and Krishnan, S. (1993). CHARM++: A Portable Concurrent Object Oriented System Based on C++. In *OOPSLA*, volume 93, pages 91–108. Citeseer.

Mattson, T., Cledat, R., Cavé, V., Sarkar, V., Budimlić, Z., Chatterjee, S., Fryman, J., Ganev, I., Knauerhase, R., Lee, M., et al. (2016a). The Open Community Runtime: A Runtime System for Extreme Scale Computing. In *High Performance Extreme Computing Conference (HPEC), 2016 IEEE*, pages 1–7. IEEE.

Mattson, T. et al. (2016b). OCR - The Open Community Runtime Interface, Version 1.2.0.

Moreland, K. (2012). Oh, $#*@! exascale! The Effect of Emerging Architectures on Scientific Discovery. In *2012 SC Companion: High-Performance Computing, Networking, Storage and Analysis (SCC)*, pages 224–231. IEEE.

OpenMP Architecture Review Board (2018). OpenMP Application Programming Interface Version 5.0.

Scratchapixel2.0 (2009-2016). A Minimal Ray-Tracer. https://www.scratchapixel.com/code.php?id=10&origin=/lessons/3d-basic-rendering/minimal-ray-tracer-rendering-simple-shapes.

Silberschatz, A., Galvin, P. B., and Gagne, G. (2008). *Operating System Concepts*. Wiley Publishing, 8th edition.

Sterling, T., Anderson, M., and Brodowicz, M. (2017). A Survey: Runtime Software Systems for High Performance Computing. *Supercomputing Frontiers and Innovations*, 4(1):48–68.

Thoman, P., Hasanov, K., Dichev, K., Iakymchuk, R., Aguilar, X., Gschwandtner, P., Lemarinier, P., Markidis, S., Jordan, H., Laure, E., et al. (2017). A Taxonomy of Task-based Technologies for High-Performance Computing. In *International Conference on Parallel Processing and Applied Mathematics*, pages 264–274. Springer.

Whitlock, B., Favre, J. M., and Meredith, J. S. (2011). Parallel In Situ Coupling of Simulation with a Fully Featured Visualization System. In *Proceedings of the 11th Eurographics Conference on Parallel Graphics and Visualization*, EGPGV '11, pages 101–109, Aire-la-Ville, Switzerland, Switzerland. Eurographics Association.

Yu, H., Wang, C., Grout, R. W., Chen, J. H., and Ma, K.-L. (2010). In Situ Visualization for Large-Scale Combustion Simulations. *IEEE Comput. Graph. Appl.*, 30(3):45–57.

# Appendix A

# Raytracer Source Code

```
#include <cstdio>
#include <cstdlib>
#include <cmath>
#include <memory>
#include <vector>
#include <utility>
#include <cstdint>
#include <iostream>
#include <fstream>
#include <cmath>
#include <limits>
#include <random>
#include <sstream>
#include <thread>
#ifdef __linux__
#include <unistd.h>
#endif

#include <ocr.h>
#include "geometry.h"
#include "trajectory.h"
#include "frame.h"


const double kInfinity = std::numeric_limits<double>::max();
std::random_device rd;
std::mt19937 gen(rd());
std::uniform_real_distribution<> dis(0, 1);

inline
```

```
double clamp(const double &lo, const double &hi, const double &v)
{ return std::max(lo, std::min(hi, v)); }

inline
double deg2rad(const double &deg)
{ return deg * M_PI / 180; }

inline
Vec3d mix(const Vec3d &a, const Vec3d& b, const double &mixValue)
{ return a * (1 - mixValue) + b * mixValue; }

struct Options
{
    uint32_t width;
    uint32_t height;
    u32 num_width_decomp;
    u32 num_height_decomp;
    double fov;
    Matrix44f cameraToWorld;
    std::string outfile;
};

enum col {red, blue, green};

// [comment]
// Object base class
// [/comment]
class Object
{
public:
    // Object() : color(dis(gen), dis(gen), dis(gen)) {}    // for random
       ↪ colors of the 'atoms'/spheres
    Object() : color(1, 0, 0) {}
    explicit Object(col col)
    {
        switch (col) {
            case red:
                this->color = Vec3d{1, 0, 0};
                return;
            case blue:
                this->color = Vec3d{0, 0, 1};
            case green:
                this->color = Vec3d{0, 1, 0};
```

```
        }

    };
    virtual ~Object() = default;
    // Method to compute the intersection of the object with a ray
    // Returns true if an intersection was found, false otherwise
    // See method implementation in children class for details
    virtual bool intersect(const Vec3d &, const Vec3d &, double &) const =
       ↪ 0;
    // Method to compute the surface data such as normal and texture
       ↪ coordnates at the intersection point.
    // See method implementation in children class for details
    virtual void getSurfaceData(const Vec3d &, Vec3d &, Vec2d &) const = 0;
    Vec3d color;

private:
    // Object& operator=(const Object&) = delete;
};

// [comment]
// Compute the roots of a quadratic equation
// [/comment]
bool solveQuadratic(const double &a, const double &b, const double &c,
   ↪ double &x0, double &x1)
{
    double discr = b * b - 4 * a * c;
    if (discr < 0) return false;
    else if (discr == 0) {
        x0 = x1 = - 0.5 * b / a;
    }
    else {
        double q = (b > 0) ?
                   -0.5 * (b + sqrt(discr)) :
                   -0.5 * (b - sqrt(discr));
        x0 = q / a;
        x1 = c / q;
    }

    return true;
}

// [comment]
// Sphere class. A sphere type object
```

```
// [/comment]
class Sphere : public Object
{
public:
    Sphere() = default;
    Sphere(const Vec3d &c, const double &r) : Object(col::red), radius(r),
       ↪ radius2(std::pow(r, 2)), center(c) {}

    // [comment]
    // Ray-sphere intersection test
    //
    // \param orig is the ray origin
    //
    // \param dir is the ray direction
    //
    // \param[out] is the distance from the ray origin to the intersection
       ↪ point
    //
    // [/comment]
    bool intersect(const Vec3d &orig, const Vec3d &dir, double &t) const
       ↪ override
    {
        double t0, t1; // solutions for t if the ray intersects
#if 0
        // geometric solution
        Vec3d L = center - orig;
        double tca = L.dotProduct(dir);
        if (tca < 0) return false;
        double d2 = L.dotProduct(L) - tca * tca;
        if (d2 > radius2) return false;
        double thc = sqrt(radius2 - d2);
        t0 = tca - thc;
        t1 = tca + thc;
#else
        // analytic solution
        Vec3d L = orig - center;
        double a = dir.dotProduct(dir);
        double b = 2 * dir.dotProduct(L);
        double c = L.dotProduct(L) - radius2;
        if (!solveQuadratic(a, b, c, t0, t1)) return false;
#endif
        if (t0 > t1) std::swap(t0, t1);
```

```
        if (t0 < 0) {
            t0 = t1; // if t0 is negative, let's use t1 instead
            if (t0 < 0) return false; // both t0 and t1 are negative
        }

        t = t0;

        return true;
    }
    // [comment]
    // Set surface data such as normal and texture coordinates at a given
       ↪ point on the surface
    //
    // \param Phit is the point ont the surface we want to get data on
    //
    // \param[out] Nhit is the normal at Phit
    //
    // \param[out] tex are the texture coordinates at Phit
    //
    // [/comment]
    void getSurfaceData(const Vec3d &Phit, Vec3d &Nhit, Vec2d &tex) const
       ↪ override
    {
        Nhit = Phit - center;
        Nhit.normalize();
        // In this particular case, the normal is simular to a point on a
           ↪ unit sphere
        // centred around the origin. We can thus use the normal coordinates
           ↪  to compute
        // the spherical coordinates of Phit.
        // atan2 returns a value in the range [-pi, pi] and we need to remap
           ↪  it to range [0, 1]
        // acosf returns a value in the range [0, pi] and we also need to
           ↪ remap it to the range [0, 1]
        tex.x = (1 + atan2(Nhit.z, Nhit.x) / M_PI) * 0.5;
        tex.y = acos(Nhit.y) / M_PI;
    }
    double radius, radius2;
    Vec3d center;
};

// [comment]
// Returns true if the ray intersects an object. The variable tNear is set
```

```
   ↪ to the closest intersection distance and hitObject
// is a pointer to the intersected object. The variable tNear is set to
   ↪ infinity and hitObject is set null if no intersection
// was found.
// [/comment]
bool trace(const Vec3d &orig, const Vec3d &dir, const std::vector<Sphere> &
   ↪ objects, double &tNear, const Sphere *&hitObject)
{
    tNear = kInfinity;
    auto iter = objects.begin();
    for (; iter != objects.end(); ++iter) {
        double t = kInfinity;
        if ((*iter).intersect(orig, dir, t) && t < tNear) {
            hitObject = &*iter;
            tNear = t;
        }
    }

    return (hitObject != nullptr);
}

// [comment]
// Compute the color at the intersection point if any (returns background
   ↪ color otherwise)
// [/comment]
Vec3d castRay(
        const Vec3d &orig, const Vec3d &dir,
        const std::vector<Sphere> &objects)
{
    Vec3d hitColor = 0;
    const Sphere* hitObject = nullptr; // this is a pointer to the hit
       ↪ object
    double t; // this is the intersection distance from the ray origin to
       ↪ the hit point
    if (trace(orig, dir, objects, t, hitObject)) {
        Vec3d Phit = orig + dir * t;
        Vec3d Nhit;
        Vec2d tex;
        hitObject->getSurfaceData(Phit, Nhit, tex);
        // Use the normal and texture coordinates to shade the hit point.
        // The normal is used to compute a simple facing ratio and the
           ↪ texture coordinate
        // to compute a basic checker board pattern
```

```
        double scale = 4;
        double pattern = (fmod(tex.x * scale, 1) > 0.5) ^ (fmod(tex.y *
            ↪ scale, 1) > 0.5);
        hitColor = std::fmax(0.f, Nhit.dotProduct(-dir)) * mix(hitObject->
            ↪ color, hitObject->color * 0.8, pattern);
    }

    return hitColor;
}

ocrGuid_t renderEdt( u32 paramc, u64* paramv, u32 depc, ocrEdtDep_t depv[]){
    ocrGuid_t optionsGuid = depv[0].guid;
    auto options = static_cast<Options*>(depv[0].ptr);
    ocrGuid_t objectsGuid = depv[1].guid;
    auto objs = static_cast<Sphere*>(depv[1].ptr);
    ocrGuid_t framebufferGuid = depv[2].guid;
    auto framebufferData = static_cast<Vec3d *>(depv[2].ptr);
    Vec3d *pix = framebufferData;

    auto w_low = static_cast<u32>(paramv[0]);
    auto w_high = static_cast<u32>(paramv[1] + 1);
    auto h_low = static_cast<u32>(paramv[2]);
    auto h_high = static_cast<u32>(paramv[3] + 1);
    auto currentIteration = static_cast<u32>(paramv[4]);
    auto num_atoms = static_cast<u32>(paramv[5]);

    std::vector<Sphere> objs_of_interest;
    for (u32 idx = 0; idx < num_atoms; idx++)
    {
        if ((objs[idx].center.z <= 5.5) && (objs[idx].center.z > 0))
        {
            objs_of_interest.push_back(objs[idx]);
        }
    }

    double scale = tan(deg2rad((options->fov * 0.5)));
    double imageAspectRatio = (double)(w_high-w_low)/(h_high - h_low);

    Vec3d orig;
    options->cameraToWorld.multVecMatrix(Vec3d(0), orig);
    for (u32 j = h_low; j < h_high; ++j) {
        for (u32 i = w_low; i < w_high; ++i) {
```

```
            double x = (2 * (i + 0.5) / (double)options->width - 1) *
               ↪ imageAspectRatio * scale;
            double y = (1 - 2 * (j + 0.5) / (double)options->height) * scale
               ↪ ;

            // https://www.scratchapixel.com/lessons/3d-basic-rendering/ray-
               ↪ tracing-generating-camera-rays/generating-camera-rays
            // generate direction vector from origin through all pixels in '
               ↪ normalized device coordinates'
            Vec3d dir;
            options->cameraToWorld.multDirMatrix(Vec3d(x, y, -1), dir);
            dir.normalize();
            Vec3d pixval = castRay(orig, dir, objs_of_interest);
            *(pix++) = pixval;
        }
    }

    ocrDbRelease(framebufferGuid);
    return NULL_GUID;
}

u32 pix2idx(u32 w, u32 h, Options& opt)
{
    u32 w_half = opt.width/opt.num_width_decomp;
    u32 h_half = opt.height/opt.num_height_decomp;
    return w/w_half + h/h_half * opt.num_height_decomp;
}

u32 pix2pixpos(u32 w, u32 h, Options& opt){
    u32 w_half = opt.width/opt.num_width_decomp;
    u32 h_half = opt.height/opt.num_height_decomp;
    u32 p_w = w % w_half;
    u32 p_h = h % h_half;
    return p_h * w_half + p_w;
}


ocrGuid_t writeImageEdt( u32 paramc, u64* paramv, u32 depc, ocrEdtDep_t depv
   ↪ []){
    ocrGuid_t optionsGuid = depv[0].guid;
    auto options = static_cast<Options*>(depv[0].ptr);

    u64 currentIteration = paramv[0];
```

```
    std::cout << "finished␣iteration␣" << currentIteration * 10 << std::endl
       ↪ ;
    ocrProgressReport(currentIteration * 10);

    ocrGuid_t sphereDbGuid = {.guid=static_cast<s64>(paramv[1])};
    u8 retVal = ocrDbDestroy(sphereDbGuid);
    if (retVal != 0) { std::cout << "failed␣to␣destroy␣Db␣of␣iteration␣" <<
       ↪ currentIteration << std::endl;}

    std::string strng;
#ifdef __linux__
    strng += "/home/ender/Sim_Results/" + options->outfile \
        + std::to_string(currentIteration) + ".ppm";
#else
    strng += "/Users/enjo/Documents/Uni/Master_Thesis/Sim_Results/" +
       ↪ options->outfile \
        + std::to_string(currentIteration) + ".ppm";
#endif
    std::ofstream ofs(strng, std::ios::out | std::ios::binary | std::ios::
       ↪ trunc);

    if (!ofs.is_open())
    {
        std::cerr << "filestream␣not␣open!" << std::endl;
        exit(1);
    }

    ofs << "P6\n" << options->width << "␣" << options->height << "\n255\n";
    u32 numFramebuffers = (depc - 1) / 2;
    ocrGuid_t framebufferGuid[numFramebuffers];
    Vec3d* framebufferPtrs[numFramebuffers];

    int idxMax = options->height * options->width / 4;
    for (int idx = 0; idx < numFramebuffers; idx++) {
        u32 depv_idx = depc - numFramebuffers + idx;
        framebufferGuid[idx] = depv[depv_idx].guid;
        framebufferPtrs[idx] = static_cast<Vec3d *>(depv[depv_idx].ptr);
    }

    char r, g, b;
    for (u32 h = 0; h < options->height; h++)
    {
        for (u32 w = 0; w < options->width; w++)
```

```
        {
            u32 id = pix2idx(w, h, *options);
            u32 p_pos = pix2pixpos(w, h, *options);

            r = (char) (255 * clamp(0, 1, framebufferPtrs[id][p_pos].x));
            g = (char) (255 * clamp(0, 1, framebufferPtrs[id][p_pos].y));
            b = (char) (255 * clamp(0, 1, framebufferPtrs[id][p_pos].z));
            ofs << r << g << b;
        }
    }

    for (auto fb_guid : framebufferGuid)
    {
       u8 res = ocrDbDestroy(fb_guid);
       if (res != 0) std::cout << "ocrDbDestroy failed" << std::endl;
    }
    ofs.close();
    return NULL_GUID;
}

ocrGuid_t finalizeEdt( u32 paramc, u64* paramv, u32 depc, ocrEdtDep_t depv
   ↪ []){
    std::cout << "finalized execution" << std::endl;
#ifdef __linux__
    std::ofstream ofs("/home/ender/Sim_Results/finish", std::ios::out | std
       ↪ ::ios::binary | std::ios::trunc);
    ofs.write("done", 4);
    ofs.close();
#endif
    ocrShutdown();
    return NULL_GUID;
}

void getObjectsFromFrame(std::vector<Sphere> *&objects, comd::Frame &frm)
{
    for (auto& atom : frm.positions())
    {
        if (((0.5 + atom[2]) * 10) <= 5.5) {
            objects->push_back(Sphere{Vec3d{(0.5 - atom[0]) * 5, (0.5 - atom
               ↪ [1]) * 5, (0.5 + atom[2]) * 10}, 0.15});
        }
    }
}
```

```
extern "C" ocrGuid_t mainEdt ( u32 paramc, u64* paramv, u32 depc,
   ↪ ocrEdtDep_t depv[]) {

    u64 argc = getArgc(depv[0].ptr);
    ocrGuid_t args;
    char** argv;
    ocrDbCreate(&args, (void**)&argv, sizeof(char*)*argc, 0, NULL_HINT,
       ↪ NO_ALLOC);
    for(u32 a = 0; a < argc; ++a) {
        std::cout << getArgv(depv[0].ptr, a) << std::endl;
        argv[a] = getArgv(depv[0].ptr, a);
    }

    Options* optionsDbPtr;

    ocrGuid_t optionsDbGuid;
    ocrDbCreate(&optionsDbGuid, (void**)&optionsDbPtr, sizeof(Options),
       ↪ DB_PROP_NONE, NULL_HINT, NO_ALLOC);

    optionsDbPtr->width = 960;// 640;
    optionsDbPtr->height = 720;// 480;

    optionsDbPtr->num_height_decomp = 4;
    optionsDbPtr->num_width_decomp = 4;
    optionsDbPtr->fov =   51.52;
    optionsDbPtr->outfile = "out";

    if( argc > 3)
    {
        optionsDbPtr->num_height_decomp = static_cast<u32>(std::stoi(argv
           ↪ [2]));
        optionsDbPtr->num_width_decomp = static_cast<u32>(std::stoi(argv[3])
           ↪ );
        std::cout << "use␣" << optionsDbPtr->num_height_decomp << "tasks␣for
           ↪ ␣height␣and␣" \
                << optionsDbPtr->num_width_decomp << "tasks␣for␣width␣
                   ↪ decomposition" << std::endl;
    }


    // unit matrix with no rotation, other possibilities at bottom of file
```

```
optionsDbPtr->cameraToWorld = Matrix44f(1, 0, 0, 0, \
 0, 1, 0, 0, \
 0, 0, -1, -1, \
 0, 0, 0, 1);

ocrDbRelease(optionsDbGuid);

gen.seed( static_cast<unsigned int>( 0 ));

u32 numTotalEdts = optionsDbPtr->num_width_decomp * optionsDbPtr->
    ↪ num_height_decomp;
u32 max_iter = 11, start_iter = 0;
if (argc > 1) {
    std::string temp{argv[1]};
    max_iter = static_cast<u32>(std::stoi(temp)/10 + 1);
}

ocrGuid_t renderTemplGuid, renderFinishEvtGuid[numTotalEdts],
    ↪ writeImageTemplGuid, \
        finalizeTemplGuid, finalizeEdtGuid, writeImageFinishEvt[max_iter
            ↪ ];

ocrEdtTemplateCreate(&renderTemplGuid, renderEdt, 6 /*paramc*/, 3 /*depc
    ↪ : options, objects, framebuffer*/);
ocrEdtTemplateCreate(&writeImageTemplGuid, writeImageEdt, 2, 1 + 2 *
    ↪ numTotalEdts /*framebufferDb[numTotalEdts] + renderFinishEvt[
    ↪ numTotalEdts] + options*/);

ocrEdtTemplateCreate(&finalizeTemplGuid, finalizeEdt, 0 /*paramc*/,
    ↪ max_iter-start_iter/*depc*/);
ocrEdtCreate(&finalizeEdtGuid, finalizeTemplGuid, EDT_PARAM_DEF, NULL,
    ↪ EDT_PARAM_DEF, NULL, EDT_PROP_NONE,
              NULL_HINT, NULL);

int w_inc = (optionsDbPtr->width / optionsDbPtr->num_width_decomp) - 1;
int w_mod = optionsDbPtr->width % optionsDbPtr->num_width_decomp;
int h_inc = optionsDbPtr->height / optionsDbPtr->num_height_decomp - 1;
int h_mod = optionsDbPtr->height % optionsDbPtr->num_height_decomp;

// iterate over frames to be processed
for(size_t iter = start_iter; iter < max_iter; ++iter) {
    u64 params[6] = {0, 0, 0, 0, 0, 0}; // w_lower, w_upper, h_lower,
        ↪ h_upper
```

```
        std::stringstream ss;
#ifdef __linux__
        ss << "/home/ender/Sim_Results/"<< optionsDbPtr->outfile << iter *
           ↪ 10 << ".xyz";
#else
        ss << "/Users/enjo/Documents/Uni/Master_Thesis/Sim_Results/"<<
           ↪ optionsDbPtr->outfile << iter * 10 << ".xyz";
#endif

        comd::Trajectory tj{ ss.str(), comd::Mode::READ, "XYZ"};
        comd::Frame frm = tj.read_step(0); // because each iteration a new
           ↪ file is written, its always step 0
        tj.close();

        // Remove file after use
        std::remove(ss.str().c_str());

        u32 numSpheres{static_cast<u32>(frm.size())};
        ocrGuid_t sphereDbGuid;

        auto* sphereDbPtr = new Sphere[numSpheres]();
        ocrDbCreate(&sphereDbGuid, (void **) &sphereDbPtr,
        sizeof(sphereDbPtr[0])*numSpheres,
                    DB_PROP_NONE, NULL_HINT, NO_ALLOC);

        u32 i = 0;
        for (auto& atom : frm.positions())
        {
            if (((0.5 + atom[2]) * 10) <= 5.5) {
                sphereDbPtr[i] = Sphere(Vec3d((0.5 - atom[0]) * 5, (0.5 -
                   ↪ atom[1]) * 5, (0.5 + atom[2]) * 10), 0.15);
            }
            else
            {
                sphereDbPtr[i] = Sphere(Vec3d(0, 0, 0), 0.15);
            }
            i++;
        }

        ocrDbRelease(sphereDbGuid);
        ocrGuid_t writeImageEdtGuid;
        u64 prm[2] = {static_cast<u64>(iter), static_cast<u64>(sphereDbGuid.
           ↪ guid)};
```

```
        params[4] = static_cast<u64>(iter);
        params[5] = static_cast<u64>(frm.size());
        ocrEdtCreate(&writeImageEdtGuid, writeImageTemplGuid, EDT_PARAM_DEF,
           ↪  prm, EDT_PARAM_DEF, NULL, EDT_PROP_NONE,
                    NULL_HINT, &writeImageFinishEvt[iter]);
        ocrAddDependence(writeImageFinishEvt[iter], finalizeEdtGuid,
           ↪ static_cast<u32>(iter - start_iter), DB_MODE_RO);
        ocrAddDependence(optionsDbGuid, writeImageEdtGuid, 0, DB_MODE_RO);

        try {
            for (int i = 0; i < optionsDbPtr->num_height_decomp; ++i) {
                params[0] = 0;
                params[1] = 0;
                if (i != optionsDbPtr->num_height_decomp - 1) {
                    params[3] = params[2] + h_inc;
                } else {
                    params[3] = params[2] + h_inc - h_mod;
                }
                for (int j = 0; j < optionsDbPtr->num_width_decomp; ++j) {

                    int idx = i * optionsDbPtr->num_height_decomp + j;
                    if (j != optionsDbPtr->num_width_decomp - 1) {
                        params[1] = params[0] + w_inc;
                    } else {
                        params[1] = params[0] + w_inc - w_mod;
                    }
                    u64 framebufferElmnts = (params[1] - params[0] + 1) * (
                       ↪ params[3] - params[2] + 1);
                    Vec3d framebufferPtr[framebufferElmnts];

                    ocrGuid_t renderEdtGuid, framebufferDbGuid;
                    ocrDbCreate(&framebufferDbGuid, (void **) &
                       ↪ framebufferPtr, framebufferElmnts * sizeof(Vec3d),
                                DB_PROP_NONE, NULL_HINT, NO_ALLOC);
                    ocrDbRelease(framebufferDbGuid);


                    ocrEdtCreate(&renderEdtGuid, renderTemplGuid,
                       ↪ EDT_PARAM_DEF, params, EDT_PARAM_DEF, NULL,
                                EDT_PROP_NONE,
                                NULL_HINT, &renderFinishEvtGuid[idx]);

                    ocrAddDependence(optionsDbGuid, renderEdtGuid, 0,
```

```
                        ↪ DB_MODE_CONST);
                    ocrAddDependence(sphereDbGuid, renderEdtGuid, 1,
                        ↪ DB_MODE_RO);
                    ocrAddDependence(framebufferDbGuid, renderEdtGuid, 2,
                        ↪ DB_MODE_RW);

                    ocrAddDependence(renderFinishEvtGuid[idx],
                        ↪ writeImageEdtGuid, idx + 1, DB_MODE_RO);
                    ocrAddDependence(framebufferDbGuid, writeImageEdtGuid,
                        ↪ numTotalEdts + idx + 1, DB_MODE_RO);

                    params[0] = params[1] + 1;
                }
                // old end-index plus one is new for next edt
                params[2] = params[3] + 1;
            }
        }catch (std::exception& e)
        {
            std::cout << "exception␣thrown␣in␣decomposition␣iteration:␣" <<
                ↪ e.what() << std::endl;
        }
        ocrEdtTemplateDestroy(renderTemplGuid);
        ocrEdtTemplateDestroy(writeImageTemplGuid);
    }
    return NULL_GUID;
}

void initializeFromFrame(std::vector<Sphere>*& objects, comd::Trajectory& tj
    ↪ , u32 step)
{
    comd::Frame frm = tj.read_step(step);
    for (auto& atom : frm.positions())
    {
        objects->push_back(Sphere{Vec3d{(0.5-atom[0])*5, (0.5-atom[1])*5,
            ↪ (0.5+atom[2])*10}, 0.15});

    }
}

void initializeRandomly(std::vector<Sphere>*& objects, u32 numSpheres)
{
    for (u32 i = 0; i < numSpheres; ++i)
    {
```

```
        Vec3d randPos((0.5 - dis(gen)) * 10, (0.5 - dis(gen)) * 10, (0.5 +
            ↪ dis(gen) * 10));
        double randRadius = 0.5;
        objects->push_back(Sphere(randPos, 1));
    }
}
```