DISSERTATION
submitted
to the
Combined Faculty for the Natural Sciences and Mathematics
of the
Ruperto-Carola University of Heidelberg, Germany
for the degree of
Doctor of Natural Sciences

Put forward by
Dipl. Inf. Dzmitry Razmyslovich
Born in Minsk, Belarus

Oral examination: 16.01.2017

Astrophysical-oriented **C**omputational multi-**A**rchitectural **F**ramework

Advisor: Prof. Dr. Reinhard Männer
Second Advisor: Dr. Guillermo Aníbal Marcus Martínez

Abstract

This work presents the framework for simplifying software development in the astrophysical simulations branch - **A**strophysical-oriented **C**omputational multi-**A**rchitectural **F**ramework (ACAF).

The astrophysical simulation problems are usually approximated with the particle systems for computational purposes. The number of particles in such approximations reaches several millions, which enforces the usage of the computer clusters for the simulations. Meanwhile, the computational extensiveness of these approximations makes it reasonable to utilize the heterogeneous clusters, using Graphics Processing Units (GPUs) and Field-Programmable Gate Arrays (FPGAs) as accelerators. At the same time, developing the programs for running on heterogeneous clusters is a complicated task requiring certain expertise in network programming and parallel programming.

The ACAF aims to simplify heterogeneous clusters programming by providing the user with the set of objects and functions covering some aspects of application developing. The ACAF targets the data-parallel problems and focuses on the problems approximated with particle systems.

The ACAF is designed as a C++ framework and is based on the hierarchy of the components, which are responsible for the different aspects of the heterogeneous cluster programming. Extending the hierarchy with new components provides the possibility to utilize the framework for other problems, other hardware, other distribution schemes and other computational methods. Being designed as a C++ framework, the ACAF keeps open the possibility to use the existing libraries and codes.

The usage example demonstrates the concept of separating the different programming aspects between the different parts of the source code. The benchmarking results reveal the execution time overhead of the program written using the framework being just 1.6% for small particle systems and approaching 0% for larger particle systems (in comparison to the bare simulation code). At the same time, the execution with different cluster configurations shows that the program performance scales almost according to the number of cluster nodes in use. These results prove the efficiency and usability of the framework implementation.

Zusammenfassung

In dieser Arbeit wird das Framework (das Programmiergerüst) für die Vereinfachung der Softwareentwicklung im Bereich der astrophysikalischen Simulationen – Astrophysikalisch-orientierte Mehrarchitektonische Rechenframework (**A**strophysical-oriented **C**omputational multi-**A**rchitectural **F**ramework (ACAF)) vorgestellt.

Die astrophysikalischen Simulationen sind normalerweise für Rechenzwecke mit den Partikelsystemen angenähert. Die Partikelanzahl bei solchen Näherungen erreicht mehrere Millionen. Dies erzwingt die Nutzung der Computercluster für die Simulationen. Wegen der Rechendichte der genäherten Simulationen werden die heterogenen Cluster mit Field Programmable Gate Arrays (FPGAs) und/oder Grafikkarten (GPUs) sehr häufig zur Beschleunigung benutzt. Die Softwareentwicklung für die heterogenen Cluster ist eine komplizierte Aufgabe, sie benötigt bestimmte Kompetenzen in der Netzwerkprogrammierung und der Parallelprogrammierung.

Das ACAF ist bestrebt, die notwendigen Programmierungsanforderungen für die heterogenen Cluster zu erleichtern. Dafür bietet das ACAF für den Benutzer einen Satz von Objekten und Funktionen an, die damit einige Sofrwareentwicklungsaspekte abdecken. Das ACAF ist auf Parallel-Daten-Probleme ausgerichtet und zielt auf die astrophysikalischen Simulationen, angenähert durch die Partikelsysteme.

Der ACAF ist entworfen als ein C++ Framework und basiert auf der Komponentenhierarchie. Diese ist verantwortlich für die verschiedenen Programmierungsaspekte für heterogene Cluster. Eine Erweiterung dieser Hierarchie mit neuen Komponenten bietet die Möglichkeit, das Framework für andere Probleme, andere Systemteile, andere Aufteilungsschemen und andere Rechenmethoden einzusetzen. Indem es als ein C++ Framework entworfen ist, gewährt das ACAF die Möglichkeit, existierende Bibliotheken und Codes zu nutzen.

Das ACAF Anwendungsbeispiel demonstriert das Trennungskonzept von verschiedenen Programmierungsaspekten zwischen verschiedenen Teilen des Quellcode. Das Benchmarking-Ergebnis zeigt, dass der Zeitzuschlag des ACAF-nutzenden Programms nur 1.6% für kleine Partikelsysteme ist und er nähert sich 0% für größere Partikelsysteme (im Vergleich zum nackten Simulationscode). Die Ausführungszeit bei verschiedenen Cluster-Konfigurationen demonstriert, dass die Programmleistung fast entsprechend der Cluster-Knoten-Anzahl skaliert. Dies weist die Effizienz und die Brauchbarkeit der Framework-Implementierung nach.

Contents

Acronyms **8**

Glossary **10**

1 Introduction **13**
1.1 Astrophysical Simulations . . . 13
1.2 N-Body Simulation Example . . . 16
1.2.1 Formal Description . . . 16
1.2.2 Computational Algorithm . . . 17
1.3 Description of the Problem . . . 22

2 Current State of The Art **24**
2.1 Standards . . . 24
2.1.1 MPI . . . 24
2.1.1.1 MVAPICH2 . . . 25
2.1.2 CUDA . . . 25
2.1.3 OpenMP . . . 26
2.1.3.1 OpenACC . . . 26
2.1.3.2 OpenHMPP . . . 27
2.1.4 OpenCL . . . 27
2.1.4.1 SyCL . . . 28
2.2 Libraries, Frameworks and Languages . . . 28
2.2.1 Cactus . . . 28
2.2.2 Charm++ . . . 29
2.2.3 Chapel . . . 30
2.2.4 Flash Code . . . 30
2.2.5 Others . . . 31

3 The Proposed Approach **33**

4 Framework Design and Implementation Aspects **35**
4.1 Why a Framework? . . . 35
4.1.1 Language Extension Approach . . . 35
4.1.2 New Programming Language Approach . . . 36
4.1.3 Framework Approach . . . 37

4.1.4 Decision Making 37
4.2 Framework Design 38
4.2.1 Target Users 38
4.2.2 Target Architecture 39
4.2.3 Three Concepts Design 39
4.2.4 Design of the Database 41
4.2.4.1 Configuration 42
4.2.4.2 Context 43
4.2.4.3 Distribution 43
4.2.4.4 Storage Objects 43
4.2.4.5 Input and Output Data Definition 43
4.2.4.6 Content Objects 44
4.2.4.7 Buffers 44
4.2.5 Design of Framework 44
4.2.5.1 Algorithm 45
4.2.5.2 Implementations 46
4.3 Design of the Framework Implementation 46
4.3.1 Device Detection Mechanism 46
4.3.2 Configuration File 48
4.3.3 Context, Database and Distribution Initialization 49
4.3.4 Content Objects and Buffers Instantiation 49
4.3.5 The Computational Concept 50
4.3.6 Simulation Execution Principles 50
4.3.7 Considered Limitations 51
4.4 Classes Description 52
4.4.1 Basic Utility Classes 52
4.4.1.1 Handle and Class 52
4.4.1.2 Logger 52
4.4.1.3 variant 52
4.4.1.4 vector_t 54
4.4.1.5 ErrorCode 54
4.4.1.6 Option 54
4.4.2 ACAF 55
4.4.3 Device 55
4.4.4 Architecture 56
4.4.4.1 CPUArchitecture 57
4.4.4.2 GPUArchitecture 57
4.4.5 Technology 57
4.4.5.1 PthreadTechnology 58
4.4.5.2 OpenCLTechnology 60
4.4.5.3 CUDATechnology 60
4.4.6 Network 61
4.4.6.1 MPINetwork 62
4.4.7 Context 62
4.4.8 Storage 62

4.4.8.1 LocalStorage and NetworkStorage . . . 63
4.4.8.2 DeviceStorage . . . 63
4.4.8.3 OpenCLStorage . . . 64
4.4.8.4 CUDAStorage . . . 64
4.4.8.5 RAMStorage . . . 64
4.4.8.6 MPIStorage . . . 64
4.4.8.7 NodeStorage . . . 64
4.4.9 Distribution . . . 65
4.4.10 Content . . . 66
4.4.10.1 LocalArray . . . 66
4.4.10.2 SyncedArray . . . 66
4.4.11 Database . . . 67
4.4.12 Kernel . . . 67
4.4.12.1 Technology::Implementation . . . 68

5 Results **69**
5.1 The Usage Example . . . 69
5.1.1 The Configuration File . . . 69
5.1.2 OpenCL Kernel Implementation . . . 70
5.1.3 pthread Kernel Implementation . . . 71
5.1.4 The Main Function . . . 72
5.1.5 Analysis . . . 74
5.1.6 Test Setup . . . 75
5.2 Benchmarking . . . 75

6 Discussion and Conclusion **79**
6.1 Pros and Cons . . . 79
6.2 Criteria Evaluation . . . 81
6.3 Retrospective . . . 82

7 Future Work **83**

Appendices **85**

A N-Body Simulation Code **86**

List of Figures **91**

Bibliography **92**

Acknowledgements **97**

Acronyms

API Application Programming Interface. 21–25, 57, 60, 66, 69, *Glossary:* API

CPU Central Processing Unit. 18, 19, 22–24, 27, 28, 35, 38, 43, 52–54, 66, 67, 71, 73, *Glossary:* CPU

CUDA Compute Unified Device Architecture. 17, 19, 22–26, 28, 35, 38, 56, 57, 60, 77, *Glossary:* CUDA

DSL Domain Specific Language. 33, 34, 78, *Glossary:* DSL

FITS Flexible Image Transport System. *Glossary:* FITS

FPGA Field-Programmable Gate Array. 12, 13, 35, 37, 38, 43, 52, *Glossary:* FPGA

GPU Graphics Processing Unit. 12, 17, 19, 22–24, 26–28, 35, 37–39, 43, 52, 53, 57, 66, 67, 71–73, 77, *Glossary:* GPU

HDF5 Hierarchical Data Format version 5. 21, *Glossary:* HDF5

MPI Message Passing Interface. 15, 16, 21, 22, 24–28, 38, 58, 60, 66, 71, *Glossary:* MPI

OOP Object-Oriented Programming. 34, *Glossary:* OOP

OpenACC Open Accelerators. 23, 24, *Glossary:* OpenACC

OpenCL Open Computing Language. 17, 19, 24–26, 35, 38, 39, 56, 60, 61, 66, 67, 73, 77, *Glossary:* OpenCL

OpenMP Open Multi-Processing. 15, 23, 24, *Glossary:* OpenMP

PCI Peripheral Component Interconnect. 43, 53, 56, 57, *Glossary:* PCI

PGAS Partitioned Global Address Space. *Glossary:* PGAS

PTX Parallel Thread Execution. 57, *Glossary:* PTX

RAM Random Access Memory. 18, 37, 39, 61, 71, *Glossary:* RAM

SCF Self-Consistent Field. 12, *Glossary:* SCF

SDK Software Development Kit. 57, *Glossary:* SDK

SPH Smoothed Particle Hydrodynamics. 12, 80, *Glossary:* SPH

SSE Streaming SIMD Extensions. 34, *Glossary:* SSE

UML Unified Modeling Language. 43, *Glossary:* UML

VGA Video Graphics Array. 43, 53, *Glossary:* VGA

Glossary

API An Application Programming Interface is a set of subroutine definitions, protocols, and tools for building software and applications. 21

CPU A Central Processing Unit is the electronic circuitry within a computer that carries out the instructions of a computer program by performing the basic arithmetic, logical, control and input/output operations specified by the instructions. 18, 35

CUDA CUDA is a parallel computing platform and application programming interface model created by Nvidia. It allows software developers and software engineers to use a CUDA-enabled graphics processing unit for general purpose processing. 17

DSL A Domain-Specific Language is a computer language specialized to a particular application domain. This is in contrast to a general-purpose language, which is broadly applicable across domains. 33

FITS Flexible Image Transport System is an open standard defining a digital file format useful for storage, transmission and processing of scientific and other images. FITS is the most commonly used digital file format in astronomy. 81

FPGA A Field-Programmable Gate Array is an integrated circuit designed to be configured by a customer or a designer after manufacturing. 12, 35

GPU A Graphics Processing Unit is a specialized electronic circuit designed to rapidly manipulate and alter memory to accelerate the creation of images in a frame buffer intended for output to a display. 12, 35

HDF5 Hierarchical Data Format is a set of file formats (HDF4, HDF5) designed to store and organize large amounts of data. 21, 80

MPI Message Passing Interface is a standardized and portable message-passing system designed by a group of researchers from academia and industry to function on a wide variety of parallel computers. 15, 21, 35, 58

OOP Object-Oriented Programming is a programming paradigm based on the concept of "objects“, which may contain data, in the form of fields, often known as attributes; and code, in the form of procedures, often known as methods. 34

OpenACC Open Accelerators is a programming standard for parallel computing developed by Cray, CAPS, Nvidia and PGI. The standard is designed to simplify parallel programming of heterogeneous CPU/GPU systems. 23

OpenCL Open Computing Language is a framework for writing programs that execute across heterogeneous platforms consisting of central processing units, graphics processing units, digital signal processors, field-programmable gate arrays and other processors or hardware accelerators. 17, 24

OpenMP Open Multi-Processing is an application programming interface that supports multi-platform shared memory multiprocessing programming in C, C++, and Fortran, on most platforms, processor architectures and operating systems, including Solaris, AIX, HP-UX, Linux, OS X, and Windows. 15, 23

PCI Conventional PCI, often shortened to PCI, is a local computer bus for attaching hardware devices in a computer. 43

PGAS A Partitioned Global Address Space is a parallel programming model. It assumes a global memory address space that is logically partitioned and a portion of it is local to each process, thread, or processing element. 37

PTX Parallel Thread Execution is a pseudo-assembly language used in Nvidia's CUDA programming environment. 57

RAM Random-Access Memory is a form of computer data storage. A random-access memory device allows data items to be read or written in almost the same amount of time irrespective of the physical location of data inside the memory. 18

SCF The Self-Consistent Field method is an algorithm for evolving collisionless stellar systems [25]. 12

SDK A Software Development Kit is typically a set of software development tools that allows the creation of applications for a certain software package, software framework, hardware platform, computer system, video game console, operating system, or similar development platform. 57

SPH Smoothed-particle hydrodynamics is a computational method used for simulating fluid flows. It was developed by Gingold and Monaghan [23] and Lucy [31] initially for astrophysical problems. 12

SSE Streaming SIMD Extensions is an SIMD instruction set extension to the x86 architecture, designed by Intel and introduced in 1999 in their Pentium III series processors. SSE contains 70 new instructions, most of which work on single precision floating point data. 34

UML The Unified Modeling Language is a general-purpose, developmental, modeling language in the field of software engineering that is intended to provide a standard way to visualize the design of a system. 43

VGA Video Graphics Array refers specifically to the display hardware first introduced with the IBM PS/2 line of computers in 1987, but through its widespread adoption has also come to mean either an analog computer display standard. 43

Chapter 1

Introduction

1.1 Astrophysical Simulations

Astrophysics is a branch of astronomy which applies physics to the study of astronomical objects, their occurrences and their physical processes. The only actual form of data which astrophysicists have comes from observation of the cosmos. In this respect, astrophysicists have no opportunity to experiment on any actual physical objects, while creating simulations both of the objects and of the environment is often a very complicated task. These complications lead astrophysicists to focus on theoretical research and to use computational simulations as a research method.

The most important computational astrophysical problems include N-Body simulations, interstellar gas simulations, cosmic dust simulations and radiative transfer simulations. For the purposes of calculation, these problems are usually approximated with respective particle systems. A particle system is a numerical approximation technique dealing with the existence and interactions of particles, which refer to some matter or radiation. The computational astrophysics data represents a collection of particles, which approximate bodies for N-Body simulations, molecules for gas simulations, dust pieces for cosmic dust simulations and photons for radiative transfer simulations. Each particle contains a number of parameters, such as position in 3D space, speed, direction, mass and other physical and/or chemical characteristics. A collection of certain values for all parameters of all particles is called the state of a particle system. At the same time, the computational tasks embrace numerical solving for a number of equations, which evaluate the state of a particle system [12].

The most critical part of utilizing particle systems for astrophysical simulations lies in time-stepping and integration techniques involved. There are many methods prescribing the certain utilization schema. The most used methods include the following:

- for N-Body simulations [25]:
    1. Particle-Particle (PP) method. The calculation consists in finding pairwise forces between all the particles and accumulating them.
    2. Particle-Mesh (PM) method. The particle system is converted into a grid ("mesh"). Then the potential is solved for this density grid and the forces are applied according to the cell assignment of the particle.

3. Particle-Particle/Particle-Mesh (P3M) method. This method is a combination of previous methods: the forces of the nearby particles are calculated by the Particle-Particle method, the influence of the distant particles is calculated by the Particle-Mesh method.
4. Nested Grid Particle-Mesh method. This method is the enhanced Particle-Mesh method which introduces the sub-grids in the cells of the parent mesh increasing the force and mass resolutions.
5. Tree-Code Top Down method. The whole particles space is iteratively split into eight octants constructing the tree (such tree is also called "octree"). For each octant the algorithm computes the total mass and the center of mass. The splitting stops, when the size of the octant, the number of particles inside or the mass of the octant reaches some threshold. The force equation is solved for the tree nodes. The forces of nearby particles are calculated by the Particle-Particle method.
6. Tree-Code Bottom Up method. This method is similar to the previous one, but the tree is built iteratively by uniting two nearest leafs at each iteration. Initially, all particles are considered to be leafs.
7. Fast-Multipole-Method (FMM) method. This method is also based on the octree technique, but instead of the forces, the potentials are calculated.
8. Tree-Code Particle-Mesh method. This method is a combination of the Tree-Code method and the Particle-Mesh method. Primarily, the particle system is converted into a grid according to the Particle-Mesh method. But the Tree-Code method is used for the grid cells, where the mass density is high.
9. Self-Consistent Field (SCF) method. This method is designed for the collisionless stellar systems, where the particles do not directly interact with one another. Rather the particles contribute to the combined gravitational field moving along orbits, such as stars in galaxies.

- for interstellar gas simulations:

1. Molecular Dynamics method [10, 21]. This method is used for studying the physical movements of atoms and molecules. The simulation consists in solving Newton's equations of motion for the particle system until its properties no longer change with time.
2. Monte-Carlo methods [39]. This is the class of the computational algorithms which use the probabilistic models for simulating some interactions and events.
3. Particle-in-cell method [42]. This method is an adoption of the N-Body simulation methods for fluid and gas simulations. In particular, the following N-Body simulation methods are used: Particle-Particle, Particle-Mesh and Particle-Particle/Particle-Mesh.
4. Smoothed Particle Hydrodynamics (SPH) method [34]. This method defines the spatial distance ("smoothing length"), over which the properties of the particles (density, temperature) are "smoothed" by a special kernel function.

- for cosmic dust and radiative transfer simulations, also Monte-Carlo method is used [36].

The particle systems discovered by astrophysicists can often contain several million particles. Consequentially, in order to perform the simulations, astrophysicists use a high level of computer power (for example, 1.7 PetaFlop/s is used for DRAGON simulations [44]).

The astrophysical simulations approximated with particle systems represent the data-parallel problems. Data parallelism is a form of the parallelization paradigm, which focuses on distributing the data across different computational units and performing the same calculation on different data simultaneously. Another form of parallelization paradigm is called task parallelism, which focuses on distributing independent calculations across the computational units.

Data parallelism and computational extensiveness have led to the greater use of the computer cluster for astrophysical simulations. Moreover, since the computational extensiveness is concentrated upon the evaluation of the state of a particle system, heterogeneous clusters become a must for an efficient astrophysical simulation [35, 19, 41, 27]. According to TOP500, the top-rated heterogeneous clusters use Graphics Processing Units (GPUs) or Field-Programmable Gate Arrays (FPGAs) as computational accelerators. In particular, there are several task-specific astrophysical projects using the GRAPE-6 FPGA for accelerating N-Body simulations [32, 28, 46, 33].

This means that astrophysicists should deal with developing simulation programs which are capable of running on heterogeneous clusters. At the same time, the development of the programs for heterogeneous clusters is a complicated task. It requires certain expertise in the following subjects:

- astrophysics - that is to say, knowledge of cosmic objects, their existence and interactions between them, including the knowledge of the physical and chemical laws involved;

- network programming - that is, knowledge of the communication between the nodes in computer clusters, including the data transfer interfaces, protocols and supporting libraries;

- parallel programming and hardware accelerators programming - that is, knowledge of the efficient data-parallel coding, the design and specifics of different computational unit architectures, the commands and libraries for communicating with hardware accelerators;

- micro-electronics for designing FPGA boards - understanding the integrated circuits of FPGA boards is necessary for choosing the correct design for the particular problem.

The following N-Body simulation example demonstrates the key programming difficulties which one encounters in working towards the development of the programs for heterogeneous clusters.

1.2 N-Body Simulation Example

1.2.1 Formal Description

The example solves a typical astrophysical force N-Body simulation problem used for simulating stellar systems. The full code of the simulation written in C++ can be found in Appendix A. Formally, the problem can be described as following:

1. the N-Body force simulation approximates the motion of particles, which interact with one another through some type of physical forces. Our usage example considers the gravitational force;

2. the problem considers N particles, which are point masses $m_i, i = \overline{1,N}$ in three dimensional space $\mathbb{R}^3$ with a position vector p_i;

3. according to the Newton's law of gravity the gravitational force felt on particle i by a single particle j is given by

$$F_{ij} = \frac{Gm_im_j(p_j - p_i)}{\|p_j - p_i\|^3}$$

   where G is the gravitational constant and $\|p_j - p_i\|$ is the magnitude of the distance between p_i and p_j;

4. summing over all the particles yields the equation of motion:

$$m_i \frac{dv_i}{dt} = \sum_{j=1, j \neq i}^{N} \frac{Gm_im_j(p_j - p_i)}{\|p_j - p_i\|^3}$$

   where v_i is the velocity of particle i and $\frac{dv_i}{dt}$ is the motion acceleration of particle i;

5. taking into account that $v_i = \frac{dp_i}{dt}$, it is possible to approximate:

$$v_i' = v_i + F_{ij}m_i\Delta t$$

$$p_i' = p_i + v_i'\Delta t$$

   where v_i' is the new velocity of particle i and p_i' is the new position of particle i.

This means that the N-Body force simulation lies in iterating over some time period with the step Δt. At each iteration the simulation consists of evolving the state of the particle system by recalculating the new positions of all the particles.

1.2.2 Computational Algorithm

The complexity of the calculation algorithm of N-Body simulation depends on the target computational configuration. Performing the calculation using the more complex configurations (a parallel calculation, a calculation in network, a heterogeneous calculation) requires additional data synchronizations, data transfers, execution synchronizations. This section covers the computational algorithms for the serial calculation, the parallel calculation on a single CPU, the parallel calculation on a CPU cluster and the parallel computation on a heterogeneous cluster using Particle-Particle method.

The serial algorithm of the simulation can be implemented within the following steps (see Figure 1.1):

1. allocate the arrays for the particle parameters: masses, positions, velocities;
2. initialize the arrays with data: either random generated values or the values loaded from data files;
3. implement the time-iterating loop with the predefined time boundaries;
4. on each iteration of the loop iterate over all the particles and for each particle perform the following steps:
   (a) compute the gravitational force felt on the current particle as a sum of the pairwise forces;
   (b) when gravitational forces for all the particles are computed, calculate the new velocity in time Δt;
   (c) using the new velocity and the current position of the particle compute the new position in time Δt.

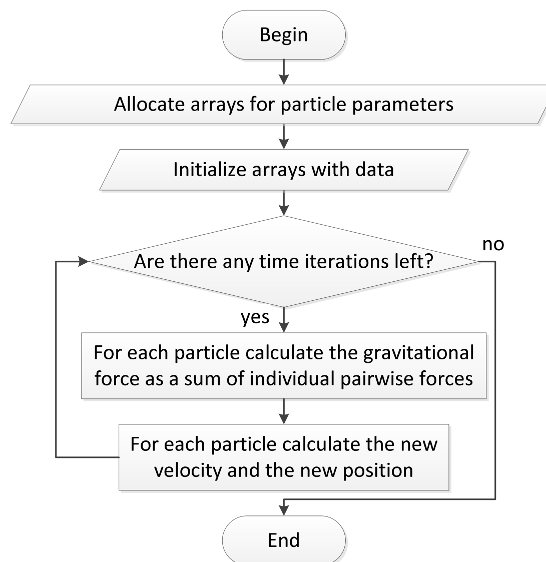

Figure 1.1: The serial algorithm of the N-Body force simulation.

The multi-threaded algorithm for a single node without any accelerators requires either the knowledge of operating systems calls to run several threads or using the third-party tools and libraries, which provide the convenience functions and instructions for these aims. The algorithm consists of the following steps (see Figure 1.2):

1. allocate the arrays for the particle parameters: masses, positions, velocities;
2. initialize the arrays with data: either random generated values or the values loaded from data files;
3. logically split the particles between the desired number of threads: decide which particles positions are computed by the particular thread - usually described by the array index range;
4. each thread performs its own time-iterating loop with the same predefined time boundaries;
5. on each iteration of the loop the thread iterates over assigned range of the particles and for each particle performs the following steps:
   (a) the thread computes the gravitational force felt on the current particle as a sum of the pairwise forces;
   (b) when gravitational forces for all the particles in all threads are computed, the thread calculates the new velocity in time Δt;
   (c) using the new velocity and the current position of the particle the thread computes the new position in time Δt;
6. at the end of each time iteration, all threads synchronize their execution by blocking the further processing till all the threads reach the end of the iteration.

The serial and multi-threaded algorithms can be developed using the regular programming language (such as C++, Java, Fortran). Using the third-party tools (such as Boost, Open Multi-Processing (OpenMP) - see Section 2.1.3) the multi-threaded version can be developed without the operating system calls knowledge.

The further optimization of the simulation code includes implementing of the distributed network-enabled algorithm. To develop the network-enabled simulation code it is necessary to have some expertise either in the network-protocol programming or in some network-messaging library (such as Message Passing Interface (MPI) - see Section 2.1.1). In case of the network-messaging library usage the algorithm includes the following steps (see Figure 1.3):

1. initialize the network library and query the network configuration (number of nodes, the rank of the current node);
2. logically split the particles between all the nodes of the network and between the desired number of threads in the context of the current node: decide which particles positions are computed by the particular node and the particular thread - usually described by the array index range;

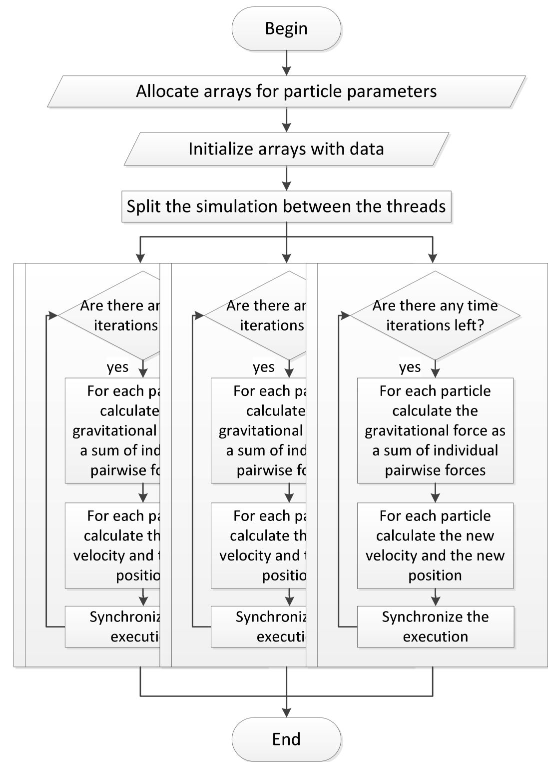

Figure 1.2: The parallel algorithm of the N-Body force simulation.

3. allocate the arrays for the particle parameters: masses, positions, velocities (velocities should be allocated only for the particles computed by the current node);

4. initialize the arrays with data: either random generated values or the values loaded from data files;

5. in case of random values the parameters should be synchronized between all the nodes (in terms of MPI library this operation is called “gathering”);

6. since the changing positions should be synchronized between the nodes at the end of each time iteration, the time-iterating loop is performed at the node-level in the main thread of the application;

7. on each iteration of the loop multiple threads are started to perform the computation of new positions of the particles, where each thread performs the same steps as previously:

   (a) the thread computes the gravitational force felt on the current particle as a sum of the pairwise forces;

   (b) when gravitational forces for all the particles in all threads of the current node are computed, the thread calculates the new velocity in time Δt;

   (c) using the new velocity and the current position of the particle the thread computes the new position in time Δt;

8. at the end of each time iteration, when all threads have finished their execution, the new positions of the particles are synchronized between all the nodes.

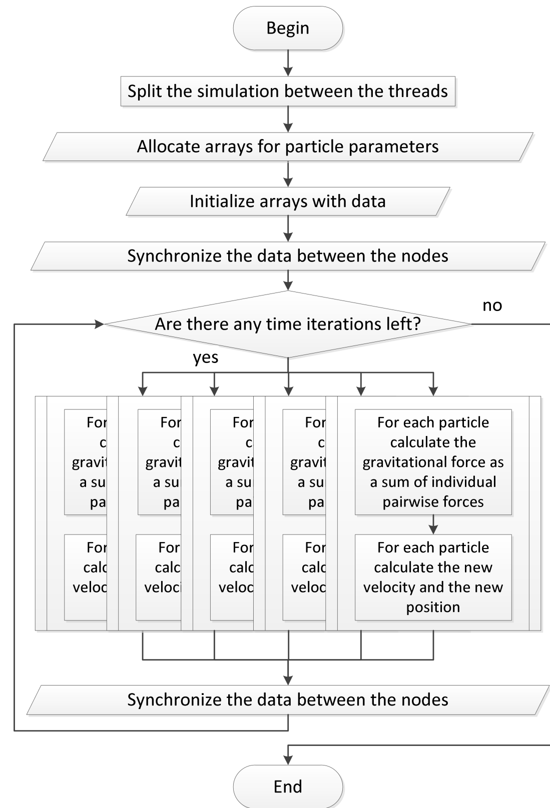

Figure 1.3: The network-enabled algorithm of the N-Body force simulation.

Finally, performing the calculation of the particle positions using some hardware accelerators requires an expertise in the accelerator programming. Particularly for utilizing GPUs, Open Computing Language (OpenCL) (see Section 2.1.4) expertise or Compute Unified Device Architecture (CUDA) (see Section 2.1.2) expertise are necessary. The algorithm in this case consists of the following steps (see Figure 1.4):

1. initialize the network library and query the network configuration (number of nodes, the rank of the current node);

2. initialize the accelerator programming environment and query the available accelerator on each node;

3. logically split the particles between all the nodes of the network and between the computational units in the context of the current node (Central Processing Unit (CPU) threads and accelerators): decide which particles positions are computed by the particular node, the particular device and the particular thread - usually described by the array index range;

4. allocate the arrays for the particle parameters for each computational unit (CPU threads use Random Access Memory (RAM) of the node, while an accelerator usually has its own memory space and can not directly communicate with RAM of the node): masses, positions, velocities (velocities should be allocated only for the particles computed by the current device);

5. initialize the arrays with data: either random generated values or the values loaded from data files;

6. in case of random values the parameters should be synchronized between all the nodes;

7. and if an accelerator has its own memory space, the data should be copied into the accelerators arrays;

8. since the changing positions should be synchronized between the nodes at the end of each time iteration, the time-iterating loop is performed at the node-level in the main thread of the application;

9. on each iteration of the loop the accelerator and multiple CPU threads are started to perform the computation of new positions of the particles, where each computational unit performs the same steps as previously:
   (a) the device computes the gravitational force felt on the current particle as a sum of the pairwise forces;
   (b) when gravitational forces for all the particles of the current device are computed, the device calculates the new velocity in time Δt;
   (c) using the new velocity and the current position of the particle the device computes the new position in time Δt;

10. at the end of each time iteration, when all devices have completed their execution, the new positions of the particles are synchronized in the following order:
    (a) the positions computed by the accelerator are transfered from the accelerator memory space to RAM of the current node;
    (b) the positions are synchronized in the network between all the nodes;
    (c) the synchronized positions are written back into the accelerator memory space.

In addition to the complexity of the algorithm, the code for computing the gravitational force, the new velocity and the new position of the particles should be developed for each computational unit separately. For example, in case of performing the computation on CPU and GPU, the code should be written in C++, C or Fortran languages for CPU threads and in OpenCL C or CUDA C languages for GPU.

Moreover, if due to the number of particles or to the number of parameters the full arrays do not fit into the accelerator memory space, the partial synchronization method should be applied. This method includes the following steps:

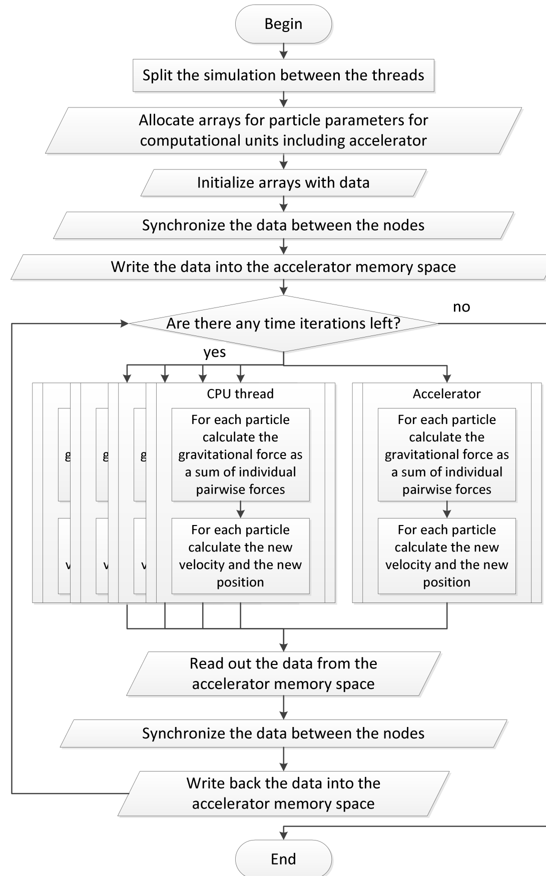

Figure 1.4: The heterogeneous algorithm of the N-Body force simulation.

1. on each time iteration split the arrays to the blocks, which fit at once into the accelerator memory space;
2. within another loop, write iteratively the blocks into the accelerator memory space and perform the computation on the individual blocks.

1.3 Description of the Problem

The development of the application, which approximates the astrophysical simulations with particle systems, is a complex task. According to the aims of the simulation, scientists should consider different methods for utilizing particle systems (see Section 1.1). For an efficient implementation of the certain method for a heterogeneous cluster, strong expertise in fields of network programming and parallel programming is required. Additionally, the developer should consider the following programming aspects (as was demonstrated in Section 1.2.2):

- the proper splitting of the problem between the nodes of the cluster and the computational devices of the nodes;

- the memory allocation for different computational units;
- the necessary and sufficient methods for synchronizing the data and the execution;
- the correct time frames and directions for transferring data;
- the correct splitting of the functionality between the computational units.

The complexity of the simulation development underlines the need for supporting tools, which address both the programming aspects and the expertise prerequisites. Hence, the supporting tools are meant to fulfil the following requirements for the efficient solution of the problem:

- the native accelerators support;
- the native network support;
- the device-specific operations abstraction;
- the network-specific operations abstraction;
- the data operations abstraction;
- the data and calculation distribution mechanisms abstraction;
- finally, the abstraction mechanisms are to keep open the option for fine-tuning the critical parts of the simulation.

Chapter 2

Current State of The Art

This chapter covers mostly used and important frameworks, libraries and standards which can optimize or simplify development of the astrophysical simulation applications.

2.1 Standards

This section gives an overview of the currently used programming standards and standard Application Programming Interfaces (APIs) for generic parallel heterogeneous programming. All the standards were designed for generic problems and therefore contain and require the implementation details which are irrelevant or obvious for the astrophysical simulation applications. Nonetheless, studying the existing standards helps to identify the relevant level of abstraction and the relevant set of functions and structures.

2.1.1 MPI

Message Passing Interface (MPI) [26] is a standardized message-passing system designed to function on a wide variety of parallel computers. MPI is widely used on many computer clusters for parallel computations on several machines. MPI can also be used for parallel computations on a single node by running multiple instances of the program. MPI provides the user with functions to efficiently exchange data in the parallel systems.

MPI has nothing to do with the code parallelization: the program can be implemented with or without accelerators usage, with or without parallelizing and optimizing execution code. MPI offers an interface-independent network communication, which enables the possibility to eliminate the usage of any particular network protocols. Moreover, MPI provides not only plain data copy functions, but also 13 collective data functions, such as data reduction, data gathering. Using of MPI the user can abstract the network communication in the efficient way.

Another advantage of MPI is the existence of a number of extensions, which often can enhance the network communication even further, such as enabling InfiniBand usage or parallel files handling (including Hierarchical Data Format version 5 (HDF5)

files). The most interesting extension in terms of heterogeneous clusters programming is MVAPICH2.

2.1.1.1 MVAPICH2

MVAPICH2 [43] is a novel MPI design which integrates CUDA-enabled GPU data movements transparently into MPI calls. This means that the user can often eliminate additional steps for transferring data firstly to the host memory and then to GPU memory and backward. Since MVAPICH2 involves not only an efficient encapsulation of the function calls, but also utilization of the motherboard and GPU chips capabilities, the final effect can reasonably improve the performance.

Hence, the user should still guarantee the correct lifetime management of GPU memory. The user should manually trigger the execution of the GPU code. If the calculations should be done simultaneously on GPU and CPU or several GPUs, MVAPICH2 can only be utilized by dividing the calculation on different devices of the same host into separate MPI processes. Another limitation of MVAPICH2 that it only supports CUDA-enabled GPUs.

So, MPI and MVAPICH2 are important libraries for developing programs for heterogeneous clusters, but these libraries cover only a single aspect - communication between nodes and devices. Moreover, the libraries actually just abstract and simplify the network protocol usage. Still, the user should take care of allocation, distribution of the data, code execution and synchronization. Therefore, these libraries can not be seen as a solution for the problem we have identified. Still, while they can be used for efficient implementing of the framework in our research.

2.1.2 CUDA

CUDA stands for and is a parallel computing platform and an API model created by Nvidia. CUDA is currently used only for Nvidia GPUs. CUDA API model enables a user to utilize GPUs for general-purpose computations on a single node.

The program written with CUDA API consists usually of 2 parts:

- a kernel code, which is going to be executed on the GPU. Usually, the kernel code represents the actual mathematical calculations, since the mathematical part is the target of the accelerators usage approach. The kernel code is written in a variety of the C language - CUDA C.

- a host code, which performs initialization, GPU memory management (including allocation, transferring and deallocation), kernel uploading and execution.

The user of CUDA API should control all aspects of the program lifetime. Having several GPUs on the same node implies explicitly controlling each device and the corresponding memory space. Performing additional parallel calculations on CPU should be implemented as a standalone solution, because CUDA has nothing to do with CPU programming.

This means that CUDA is a utility for general-purpose GPU programming. It provides a possibility to efficiently develop general-purpose programs for Nvidia GPU

devices. But being a generic tool implies that CUDA offers the user as much programming aspects as possible and requires as many implementation details as it is actually necessary. So, CUDA should be a part of the solution in our research. Still, it has not been designed to abstract the aspects we need to be covered.

2.1.3 OpenMP

Open Multi-Processing (OpenMP) [15] is a standard API for shared-memory programming in C/C++/Fortran languages, which enables easy and efficient development of the parallelized code using compiler directives. OpenMP parallelizes the program by distributing the execution of some code in the threads pool. OpenMP API is system independent, while the compiler is responsible for the correct system-dependent implementation.

In order to run some piece of code in parallel, the user should mark this code with an OpenMP pragma, it could be either a for loop, iterations of which will be distributed, or a number of sections each of which will run in a single thread. The necessary initialization calls and actual multi-threaded calls will be placed by the compiler preprocessor. Additionally, the user can specify which data should be local for a thread, which data should be shared between threads (also, some other basic data operations are available such as scattering, gathering and reduction).

Using OpenMP pragma instructions, it becomes very easy to parallelize the code for multi-threaded execution. Often, if a loop has no data dependencies between iterations, it is enough just to place a single pragma before loop and recompile the program. At the same time, parallelization of a complex code requires certain mastering in OpenMP programming, but it is usually much easier to use OpenMP for pure calculations rather than to use the thread management system-dependent calls.

The current widely supported version 3.1 (Microsoft Visual Studio 2008-2015 support only version 2.0) is designed to execute the parallel parts of the code using only CPUs. This limits the actual profit of using OpenMP as a solution for the identified problem. But there are several extensions of OpenMP which enable also accelerators usage. These extensions will be described in the following subsections.

2.1.3.1 OpenACC

Open Accelerators (OpenACC) is a standard for the programming of computational accelerators originally proposed by Nvidia (currently only CUDA-enabled GPUs are supported). Open Accelerators (OpenACC) uses the similar API as OpenMP and is also based on the preprocessor pragmas. In addition to the standard OpenMP pragmas, OpenACC offers the instructions to control data allocation, data flow, accelerator kernels and accelerator parallel blocks. Using OpenACC in case of independent loop iterations the programming of computational accelerators can be done by adding a single pragma to the code. If no accelerators are present in the machine, CPU will be used for executing the code. In 2013, OpenACC was merged into the general OpenMP standard - OpenMP version 4.0. OpenACC as well as OpenMP 4.0 are currently supported by a limited number of compilers.

2.1.3.2 OpenHMPP

OpenHMPP (HMPP for Hybrid Multicore Parallel Programming) is a programming standard for heterogeneous computing based on HMPP API developed by CAPS Enterprise. This API also uses preprocessor compiler directives for marking the code to run it on the hardware accelerator. The basic idea of OpenHMPP lies in defining a codelet - a pure calculation function which is intended to be performed by the hardware accelerator. Additionally, the user should define the data transfer points and codelet call points. At the current moment OpenHMPP is supported only by 2 compilers: CAPS Enterprise compilers and PathScale ENZO compiler suite.

Unfortunately, OpenMP and its extensions do not solve the problem as well. Even taking into account that such API model definitely reduce the requirements in parallel programming skills abstracting the numerous function calls in easy-readable pragmas, it does not hide the implementation details, which are out-of-interests for scientific programmers: device data allocation, data transferring, runtime synchronization. On the other hand, the API hides the device selection possibilities, which may be necessary for the advanced programmers. Additionally, this API does not cover at all any kind of network communications and is designed solely for a single node. This means that for heterogeneous computing clusters a user should manually manage MPI (or other) calls mixing them with OpenMP (OpenACC or OpenHMPP) pragmas to run the application on all the nodes, which furthermore complicates the final code.

2.1.4 OpenCL

Open Computing Language (OpenCL) [3] is an open standard for general purpose parallel programming across different heterogeneous processing platforms: CPUs, GPUs and others. The OpenCL programming model is quite similar to CUDA, but implies the usage aspects of different accelerators. As well as for CUDA, OpenCL program consists of 2 parts:

- a kernel code, which is going to be executed on accelerators written in OpenCL C language.
- a host code, which performs initialization, memory management (including allocation, transferring and deallocation), kernel compilation, uploading and execution.

As well as for CUDA, using OpenCL requires the user to control all aspects of the program lifetime. But in contrast to CUDA, OpenCL provides a possibility to run the kernel code on different GPUs and other different accelerators such as DSPs (Digital Signal Processors), FPGAs (Field-Programmable Gate Arrays). Also, OpenCL offers a simplified memory model for multi-accelerator contexts.

Nevertheless, OpenCL is designed for single machine implementations. There were some projects (such as CLara [1] - the project is stalled at OpenCL 1.0, or VirtualCL [7] - the project is stalled at OpenCL 1.1) which implement a proxy for the remote devices providing an access to them over the network. This implies that the host code of a proxy is not able to provide some logic, to store some temporary buffers, to optimize the network data exchange. Rather, the project implies working with remote devices

the same way as with local devices, which can lead to the unnecessarily frequent and time-expensive data transfers. This limits the utilization of OpenCL for heterogeneous clusters programming. Still, the OpenCL use is possible in conjunction with MPI or other communication interface.

Moreover, OpenCL is a standard for parallel programming of computational accelerators. This means that OpenCL is not supposed to simplify the accelerators programming (still it fulfils this task for some platforms). Instead, it provides a standard way to incorporate the accelerators power into the end-user applications. Therefore, OpenCL could be an important part of the solution for our problem.

2.1.4.1 SyCL

SyCL [4] is a new C++ single-source heterogeneous programming model for OpenCL. SyCL takes an advantage of C++11 features such as lambda functions and templates. SyCL provides high level programming abstraction for OpenCL 1.2 and OpenCL 2.2. This means that SyCL simplifies the integration of OpenCL into the programming code, making the heterogeneous programming available without learning some specific language extensions (such as the OpenCL C language or the CUDA C language). Moreover, SyCL tends to be included in the upcoming C++17 Parallel STL standard. Still, being an enhancement of OpenCL standard, SyCL does not introduce any network interoperability restricting the heterogeneous programming model to a single machine.

2.2 Libraries, Frameworks and Languages

This sections covers numerous libraries and frameworks used or possible to be used for solving the code complexity problem of heterogeneous applications. [17]

2.2.1 Cactus

Cactus [24] is an open-source modular environment, which enables parallel computation across different architectures. Modules in Cactus called “thorns”. A thorn encapsulates all user-defined code. A user has a choice either to combine the solution of the problem configuring one or several existing thorns or write a new thorn. Thorns are able to communicate with each other using the predefined API functions. A thorn consists of at least a folder and 4 administrative files written in Cactus Configuration Language: interface.ccl, param.ccl, schedule.ccl, configuration.ccl. Each of these files describes some particular properties of the configuration:

- interface.ccl is similar to a C++ class definition providing the key implementation features of the thorn;
- param.ccl tips which data is necessary for running the thorn and which data is provided by the thorn;
- schedule.ccl defines under which circumstances the thorn is executed;

- configuration.ccl specifies which milestones are required to run the thorn and which milestone provides the thorn. In the built configuration, each milestone can be provided not more than once, while the code base can have several thorns providing the same milestone. The example milestones are: LAPACK, OpenCL, IOUtil.

The rest of the thorn implementation should be organized into the files written with the following languages: Fortran90, C, C++, CUDA C, OpenCL C. The thorn implementation should include the functions defined in interface.ccl. The files will be compiled and linked together during the building of particular configuration. In functions and kernels a user should explicitly utilize the predefined Cactus macros and instructions, which are to be replaced with the necessary language constructions before compiling the program.

Cactus code has a built-in support of MPI. The latest version of Cactus includes the thorns for utilizing accelerators with the help of CUDA and OpenCL. Having these thorns, the user is able to program the accelerators calling the simplified interface functions for copying data and executing kernels. The network communication and the data transfer with accelerators can also be implicitly managed by Cactus using the distributed data types.

Nevertheless, having the distributed data types does not solve the problem completely, because implementing a new thorn is quite a complicated task. The user has no ability to combine different devices into the same solution, since the thorns are always synchronized. Cactus is only designed to solve time iterative problems.

2.2.2 Charm++

Charm++ [29] is a message-driven parallel language implemented as a C++ library. The usual Charm++ program consists of a set of objects called “chares”. A chare is an atomic function, which performs some calculations. Chares communicate with each other using messages. The task of the programmer in Charm++ context lies in dividing the problem into work pieces, which can be executed with virtual processors. The Charm++ library schedules these work pieces among the available processing units.

A chare implementation should be written in C++ language. It represents several classes which inherit some Charm++ classes. A typical chare has at least 2 classes: the main chare class, which initializes the environment and sets the necessary variables, and a worker class, which contains calculation routines. Since the source code of chares is written in C++ language, it is possible to use any 3rd party libraries including the accelerating libraries such as CUDA and OpenCL. But using these libraries anyway involves the manual management of all the aspects of accelerators programming.

Another possibility to utilize GPUs lies in using an additional Charm++ library - Charm++ GPU Manager. This library provides the user with simplified functions to interact with CUDA-enabled GPUs. The user should define a work request for GPU Manager providing a CUDA kernel, input and output arguments to be transferred to the GPU. The GPU Manager ensures the overlapping of transfers and executions on the GPU and runs a GPU kernel asynchronously.

Even with the help of the GPU Manager, writing GPU-enabled programs with Charm++ remains a complex task. Charm++ is a message-driven platform, therefore the user should program the chares keeping in mind all the possible input and output messages. A user should control all the aspects of GPU programming. With a help of GPU Manager, a user can save on some function calls. Still, he should fully control the workflow.

2.2.3 Chapel

Chapel [13] is a parallel programming language. Chapel provides a user with a high-level parallel programming model which supports data parallelism, task parallelism and nested parallelism. Being designed as a new standalone language, Chapel allows to use a high level of parallelism abstraction. This results in a compactly written code which is at the same time highly optimized, since the compiler controls all the aspects. Chapel was initially designed for multi-core Cray machines. But thanks to the high level of abstraction, Chapel was extended to support also the heterogeneous systems.

At the same time, being a standalone language, Chapel has limited possibilities for extending the functionality and for interoperating with other languages. Since Chapel is an open source project, everybody can change the compiler grammar for having new commands. Additionally, Chapel provides interoperability with the C language, which consists of implementing special binary bindings. Also, Chapel is able to generate a C interface and compile the source code into the shared library, so the code written in Chapel can be called from other C programs.

Hence, Chapel is a powerful language, which allows a user to write parallel programs with several lines of code. Since the compiler is responsible for all the aspects of deploying a parallel program: data transferring, network communication, load balancing and device's execution calls, it becomes difficult to control the workflow of the program. Moreover all the optimizations and extensions should be done on the language grammar level, which involves even higher expertise in parallel computing.

2.2.4 Flash Code

Flash Code [18, 22] is a modular Fortran90 framework targeted to computer clusters. It uses MPI to distribute calculations over the cluster nodes and inside the node over CPU cores. The Flash Code was initially developed for simulating thermonuclear flashes. But due to the modularity of the system, many other modules were implemented, which led to wider application range. The current version of Flash Code has a huge delivered code base: ca. 3500 Fortran files.

The Flash Code was designed much earlier than heterogeneous clusters became widely-used. Therefore, the framework has no built-in support for any hardware accelerators and relies on particular modules to optimize the calculations as much as possible. Flash Code has different module types. Each module type is responsible for one or another system aspect being usually quite atomic (solvers, grids, etc). This means that a module can be implemented using any accelerating techniques and libraries. Moreover, a module can be implemented as a standalone dynamic library with the necessary Fortran90 bindings to the Flash code.

But the modularity of the framework implies the unnecessary data transferring in case of heterogeneous systems. The framework can not consider the device memory, therefore, data should always be loaded into the device on the entry of the module and unloaded on the exit, even if the next module needs it to be in the device memory. Moreover, the constant variables and arrays should be transferred to the device at each iteration. These disadvantages can impair the performance gap achieved by using heterogeneous systems.

Moreover, the modularity of the Flash Code does not incorporate the abstraction of the parallel programming. So writing a new module requires the proficiency in parallel programming, including: hardware accelerators utilization; MPI usage; data distribution and synchronization techniques.

2.2.5 Others

AMUSE [47] is a Python framework designed to couple existing libraries for performing astrophysical simulations involving different physical domains and scales. The framework uses MPI to involve cluster nodes. Still, the utilization of any hardware accelerators should be a part of libraries coupled in a particular configuration.

Swarm [16] is a CUDA library for parallel n-body integrations with a focus on simulations of planetary systems. The Swarm framework targets single machines with Nvidia GPUs as hardware accelerators. The framework provides a user with a possibility to extend the calculations algorithm. But the final system is not scalable and cannot utilize the power of a cluster. So, the framework is only designed to solve some specific problems.

The **Enzo** [2] project is a adaptive mesh refinement simulation code developed by a community. The code is modular and can be extended by users. Enzo does not support network communication. Still, it contains several modules developed to utilize Nvidia GPUs using CUDA.

There are also numerous task-specific projects, which target certain astrophysical problems. In particular, the following projects should be mentioned as widely used:

- **GADGET-2** [40] is a freely available code for N-Body and gas simulations on computer clusters, which uses Tree-Code Particle Mesh and SPH methods. The older versions of the project have also supported GRAPE [32] FPGAs as computational accelerators.

- **NBODY6++GPU** [45] is a code for N-Body simulations using Ahmad-Cohen neighbor schema [8]. The code uses hybrid parallelization methods (MPI, GPU, OpenMP and Streaming SIMD Extensions (SSE)) to accelerate the simulations.

- **Vine** [46] is a simulation code for solving N-Body and interstellar gas problems using tree structures and SPH method. The code is able also to utilize GRAPE [32] FPGAs to accelerate the computation.

Among other languages which are not so widely used we should mention: **Julia** [11] language, **X10** [14] language, **Fortress** language. All these languages were initially designed for CPU clusters. Some of them provide ports or extensions for hardware

accelerators. These ports and extensions usually have no abstraction for the accelerator memory space communications.

Finally, some other widely-used, but very domain-specific libraries are: **WaLBerla [20]**, **RooFit** [9], **MLFit** [30].

Summary

The described standards, libraries, frameworks and languages solve the individual problems of the development of the astrophysical simulation programs. They are able to optimize and/or simplify the development. Still, they do not solve completely the given problem.

Chapter 3

The Proposed Approach

As described in Chapter 1, it is a complex problem to develop the heterogeneous-enabled particle system simulation application without special tools and frameworks, which in turn requires knowledge in several subjects. This takes much time and demands professional expertise from astrophysicists, which restricts scientists to perform calculation experiments on clusters easily and distracts them from the main goal.

As it was shown in Chapter 2, there are currently several different solutions, which can optimize and simplify the process of development. Each of these solutions has its pros and cons. We firstly want to highlight the importance of combining the ideas and targeting a particular kind of problem - particle system problems for astrophysical simulations. Limiting the application purposes of the solution will result in better abstractions for the necessary entities. Secondly, by designing the solution to be heterogeneous-enabled, it becomes possible to encapsulate all the necessary device-specific function calls. Finally, targeting the particle system problems enables a prospective ability to use the framework not only for astrophysical simulations, but also for other particle system domains - for instance physics and biology.

The aim of our research is to **simplify software development for the implementation of astrophysical simulations** by means of reducing programming knowledge requirements. The solution we suggest for the identified problem is the ACAF. ACAF stands for **A**strophysical-oriented **C**omputational multi-**A**rchitectural **F**ramework. The ACAF is a toolkit for developing the astrophysical simulation applications. The target data to be processed with the ACAF is a set of states of a particle system.

The ACAF aims to facilitate astrophysical research by providing a user with a set of objects and functions which fulfil the following requirements:

- the structure of an object and the semantics of a function should be plain and similar to the objects often used by scientists in other programming environments and in theoretical problem descriptions;
- the objects and functions should cover most of the heterogeneous programming aspects;
- there should be a possibility to extend the tools in use as well as to provide the alternative implementations of existing tools;

- the design of the framework should clearly split the algorithmic (mathematical, physical) part from the heterogeneous programming techniques;
- the definition of the distribution of data and computation over the cluster nodes should be user-friendly;
- the programming language for the framework implementation should be flexible enough to fulfil the previous requirements, while the language should have as little run-time expenses (e.g. by using Virtual Machines) as possible. Ideally the language should be similar to the used by scientists at the present moment;
- finally, it would be an additional advantage to maintain the possibility of continually reusing the existing computational libraries.

For evaluating the success of the design and implementation of the proposed approach, it is helpful to elaborate upon the following criteria:

1. the time overhead of using the framework in comparison to the bare simulation code: the less the time overhead is, the more successful the framework is;
2. the amount of the device-specific code (device queries, device memory allocations, device data transferring, device communication) necessary for executing a simulation in comparison to the bare simulation code. That is to say, the less the amount of code is, the more successful the framework is;
3. the amount of the network-specific code (network queries, network data transferring) necessary for executing a simulation in comparison to the bare simulation code. That is, the less the amount of code is, the more successful the framework is;
4. the portability and the flexibility of the resulting simulation in the context of deploying the simulation code within different hardware configurations. The less actions should be done for deploying the simulation code on other hardware, the more successful the framework is;
5. the uniformity of the computational code for different target devices - the less different computational codes should be written for different devices, the more successful the framework is;
6. the possibility that the present framework can be extended: the more modular and atomic the framework entities are, the more successful the framework is.

The intermediate results of the research were published in:

1. D. Razmyslovich, G. Marcus, and R. Männer. Towards an Astrophysical-oriented Computational multi-Architectural Framework. In *Computational World 2016*, pages 16 – 26. IARIA, 2016. ISSN 2308-4170
2. D. Razmyslovich and G. Marcus. Astrophysical-oriented Computational multi-Architectural Framework: Design and Implementation. *International Journal On Advances in Intelligent Systems*, volume 9(3&4), forthcoming

Chapter 4

Framework Design and Implementation Aspects

4.1 Why a Framework?

The problem described in Chapter 1 can be solved in general terms with one of the following possibilities:

- an extension for the existing compiler, which introduces some meta-commands (or pragmas in the terms of the C language), which are consequently translated into the usual language instructions and are compiled normally;
- a new programming language with a full-functional compiler;
- a framework, which provides a user with some abstraction elements, which can be combined for solving the problem.

The pros and cons of those approaches are described in the following subsections.

4.1.1 Language Extension Approach

The main advantage of the language extension is the maintenance of the programming language in use. This approach introduces some new commands, which encapsulate the existing instructions. Ideally, ignoring the new commands should not prevent the source code to be compiled.

Since the extension is triggered during the program compilation, it needs no extra run-time expenses and is equivalent to the similar hand-written code. This also means that a user can choose between the extension capabilities and hand-writing at any place in the code. Additionally, just extending the programming language voids a need to learn something completely new. Even more, the existing code can then be cheaply adapted to utilize the extension capabilities.

On the other hand, a pure language extension without any additional libraries limits the actual possibilities, because the extension is a compile-time tool. So the extension cannot have any run-time information: the amount of available resources; the devices in use; cluster utilization schema; the amount of data to be processed in non-constant

cases. This means that a user should embed all the necessary information into the source code, so that the extension has an access to it. It directly involves the recompilation of the program for any resource changes. Moreover, it means that the program should have different source code for different machines and that the binaries are not portable. Last but not least, the source code enhanced with a language extension is more difficult to debug.

A good example of the language extension is OpenACC (see Section 2.1.3.1). OpenACC enables a possibility to utilize the accelerators using the preprocessing commands. Using these commands the user marks which parts of the code should be executed on the accelerator, which data is input and which data is output. The commands are accelerator-independent. Still one of the compiler arguments is the target accelerator. This means that the final binary is compatible and optimized for some particular accelerator or several accelerators (for example, all Nvidia GPUs with compute-capability 2.0). So, the binary is not easily portable and requires the recompilation for different accelerators.

4.1.2 New Programming Language Approach

In contrast to the language extension, which introduces the new commands keeping the forms and syntax of the original language, the new programming language enables an opportunity to produce the forms and syntax according to someone's needs. This means that the language extension is bound to the expressions and patterns defined in the original language. Conversely, the new programming language is designed for solving certain problems and defines the optimal expressions for these aims.

Having a separate language designed initially to express data parallelism for some domain-specific tasks provides users with a clear problem-related set of instructions. This set of instructions can enhance the development serving the following advantages:

- the improved readability of the code;
- the advanced forms of expressions, which are convoluted in other languages;
- the development patterns promoted by the instructions;
- the higher level of abstraction;
- the necessary level of the programming flexibility.

Still, developing a new programming language is quite a complicated task which requires considering the following issues:

- granularity of the instructions/statements - which details may be interesting for users and which details are exhausted;
- extendability of the programming language contradicts the portability of the extended solution. Since the extension should be a part of the language, it should expand the semantics, which complicates the combining of several extensions;

- compatibility with the existing solutions - it is often more efficient to reuse some existing library, rather than to implement a complete solution of the problem from scratch.

Finally, the disadvantage of this approach makes the fact that the user has to learn a completely new programming language.

4.1.3 Framework Approach

In contrast to the language extension and the new programming language, the framework does not introduce new language commands, forms and expressions. Contrarily, a framework is generally language-independent and often provides the interfaces for several programming languages at the same time.

The framework provides users with an abstraction mechanism usually represented with a set of objects and functions. The objects and functions encapsulate some functionality providing an interface which fits the actual computational problem better. A successful design of the framework can lead to the softening of the dependencies between the objects and functions and their modularity. The latter means that the framework can be divided into parts and implemented and deployed part by part, which guarantees a faster feedback and results. Additionally, the modularity can secure extending possibilities. Moreover, the framework can become a base for another framework.

The framework can be usually extended to include some compile-time tools, which will further reduce the amount of the user code. Finally, it is a common practice to use a framework as a base for some Domain Specific Language (DSL).

The key advantage of the framework approach is the flexibility of its implementation and usage. At the same time, the main disadvantages of the framework approach lies in the fact that the original programming language is kept. This means that not all instructions can be abstracted and encapsulated. Moreover, the framework using requires some initialization and setting up steps. Being a run-time tool, a framework can also have additional resource expenses (memory usage, execution time).

4.1.4 Decision Making

Considering the given problem (see Section 1.3) and the simulation example (see Section 1.2), the framework has been chosen as the most efficient way to proceed due to the following reasons:

1. the programming aspects which compose the problem represent the functional complexity of the heterogeneous programming. Consequently, the framework aims to to encapsulate the complex functionality providing an appropriate interface;

2. at the same time, the programming language commands, forms and expressions do not compose any problem, because astrophysicists are used to use them;

3. the architecture of computational clusters is changing quite rapidly introducing new computational accelerators, new node interconnections, new types of nodes. Therefore, the possibilities to extend the framework are becoming very important;

4. the prospective ability to use the framework as a base for a DSL is essential, since DSL can provide a further simplification of the development;
5. finally, the project is developed by a single person. So in order to guarantee the appropriate velocity of the solution implementation, the modularity of the approach is highly important.

4.2 Framework Design

We have chosen to design and implement the proposed framework as a C++ framework. C++ combines the best options for our requirements:

- being the OS programming language for the target systems, it provides the best performance possibilities - there are options to use direct processor commands, optimized SSE commands, as well as different acceleration options: multi-threading, accelerating hardware calls;
- an additional advantage of the OS programming language lies in the possibility to link against any other binaries, so that the existing libraries (either technology-related or astrophysics-related) can be reused in the code scope;
- C++ is an Object-Oriented Programming (OOP) language, which allows to design the framework based on the objects paradigm, while the properties of Object-Oriented Programming (OOP) (encapsulation, inheritance and polymorphism) guarantee the extendability of the framework design;
- C++ namespaces provide a possibility to prevent naming collisions between different extensions and modules without limiting the extendability;
- the latest standards for C++ (C++11 and C++14) define the ability to integrate functional programming paradigm into the object-oriented environment. The functional programming enables the usage of functional data types and functional data combinators. These features can be utilized for representing complex data dependencies and data transforming functors.

4.2.1 Target Users

The design of the framework should be understandable for the following groups of users:

- Astrophysicists, as the main target users group. They need the understandable abstraction - the structures and functions similar to the native mathematical and physical mechanisms used for solving the simulation problems. At the same time, the coverage of the heterogeneous programming aspects from the structures and functions should be comprehensive in order to leave primarily the astrophysics-related parts to be programmed. Still, astrophysicists are known to be familiar with the programming languages. In particular, Fortran90 is the most used programming language among them. Therefore, using C++ for the framework makes

it available to utilize the existing libraries and the code written in Fortran90. Additionally, the user code should be stable to the framework growth and upgrades including technological upgrades.

- Programmers, who will extend the framework. The main directions in which the framework should be extendable: computational architectures and technologies (e.g. new devices and new interfaces), scaling possibilities, data handling, algorithmic enhancing. Meanwhile the main issues to deal with are the coexistence of different extensions, flexibility in extending as many entities as possible.

- Other scientists working with particle system simulation problems. Considering this group of users, the framework should be designed for the generic problems of the particle system simulation, rather than to be limited to the astrophysics-specific aspects.

4.2.2 Target Architecture

The current framework implementation is targeted to the heterogeneous clusters using Central Processing Units (CPUs), Graphics Processing Units (GPUs) and Field-Programmable Gate Arrays (FPGAs) for the computations. The nodes is to be interconnected in any way supported by Message Passing Interface (MPI), e.g. Ethernet, Infiniband. The current implementation is limited to be compiled and executed on the machines running some Linux operating system. For the compilation and executing the framework based applications the following additional packages are necessary:

- an appropriate GCC compiler version (greater than 4.7) and the corresponding standard library package;
- libpci-dev;
- libconfig++-dev;
- OpenMPI or MPICH for utilizing distributed computations;
- CUDA for utilizing CUDA technology for NVIDIA GPUs;
- OpenCL for utilizing OpenCL technology for GPUs, FPGAs and CPUs.

4.2.3 Three Concepts Design

Taking into account the mentioned target architecture and the target user groups, we have designed the framework splitting the heterogeneous programming problem into 3 concepts (see Figure 4.1):

1. The **computational concept** describes the principal algorithm used for calculating. In other words, the computational concept is a mathematical, physical and astrophysical background of the problem solution and the environment necessary to execute the solution of the problem on some particular device. This concept bases on a set of efficient high-parallel multi-architectural algorithms. So

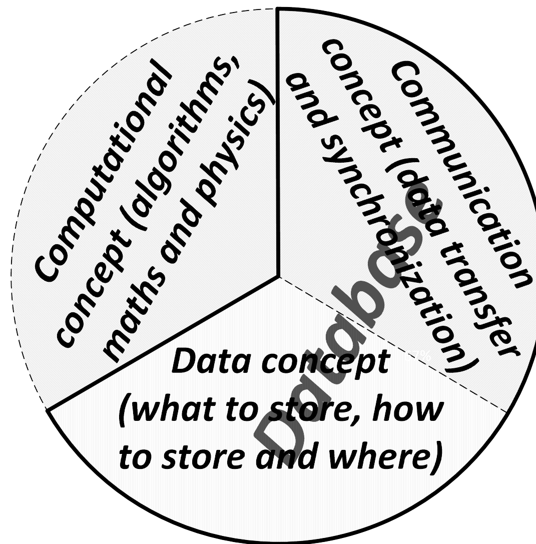

Figure 4.1: The 3-concepts design.

that each algorithm had an efficient implementation for each architecture and device in use. All implementations for the same algorithm could work together on different platforms.

2. The **data concept** describes logical and physical representation of the data used in a solution, as well as the distribution of this data. This concept lies both in a set of data-structures providing an efficient way of managing the data of the astrophysical objects; and a set of functions for manipulating these structures.

3. The **communication concept** describes data transfers and synchronization points between computing units. The concept lies in efficient data-distribution mechanisms, which guarantee the presence of the necessary data in the required memory space and in the required order. This means that the communication concept is responsible for transferring data from one memory space to another and for transforming it according to the user-defined, architecture-defined or device-defined rules.

Design of the computational concept is a technical problem lying in the space of a properly implemented set of programming interfaces to access the necessary functions on the necessary platforms.

Conversely, the design of the data concept and the communication concept can be coupled into a special distributed database. Here and further, we understand under the database its basic definition: a database is an organized collection of data. This database should provide the user with an interface for managing data. Besides, it should manipulate the data according to the requirements and properties of computational units and algorithms. Hence, the database should fulfil the following requirements:

- operating with a set of structures efficient for representing astrophysical data: tuples, trees (octrees, k-d trees), arrays;

- operating with the large amount of data, which does not fit in the target device memory;
- the native support of hardware accelerators e.g. GPUs and FPGAs;
- the data should be efficiently distributed between both cluster nodes and the calculating devices inside of each node;
- the database should be programmatically scalable: a user should be able to extend the number of features in use - architectures; devices; data-structures; data manipulation schemes and functions; communication protocols;
- the database should store the data according to the algorithm, device and platform requirements.

This means that this special database can be referred to as a Partitioned Global Address Space (PGAS), which is already addressed in several existing solutions such as Chapel and X10. But in our approach, we incorporate into the database not only partitioning of the address space, but also the other properties specified above.

Hence in this work, we focus on the communication and data concepts - **the design and implementation of a distributed database**. The computational concept is designed to contain only the algorithms and functions, necessary to present the capabilities of the database.

4.2.4 Design of the Database

The target data for the ACAF database is a set of states of a particle system. According to the definition of a particle system (see Chapter 1), there is no need for our database to store various data of various types. All parameters of a particle are some physical properties of it. So in computer representation, the parameters are usually either integer, float or double (integral) values. Hence in our database, these types of data are only considered. A state of some particle system can be represented in some computer memory space as an array of structures, where members of a structure are particle parameters - e.g. integral data types. Therefore, the ACAF database is only targeted to store arrays of integral data elements.

As soon as a particle system usually includes some million particles, it is common and necessary to use computer clusters and accelerators to simulate its states. So, the aim of the ACAF is to simplify implementing the simulation tasks targeted to be run on heterogeneous computer clusters utilizing as much computational power as possible. The efficient utilization of any computational device (e.g. a processing unit) becomes possible only when all the parameters necessary for computation reside in the cheapest memory space in terms of access latency. The efficient use of low-level memory spaces (processor registers and near by caches of the unit) is to be achieved with both the compiler implementation and the operating system scheduler. Meanwhile, the programmer's task is to ensure the presence of data in the nearest high-level memory space (usually the device RAM). Moreover it is necessary to store data in high-level memory spaces in the format acceptable for computational algorithms. Hence, raw

arrays are maintained in our database. This provides the direct access to the parameters of a particle.

The ability of the ACAF database to distribute data between cluster nodes and devices enables the scalability of the data amount. So, the amount of data to be processed is only limited to the mutual storage capabilities of cluster nodes and devices.

Distributing data between cluster nodes and devices implies division and synchronization of data according to the particular implementation of the computational concept. Meanwhile, data synchronization in heterogeneous computer clusters implies interoperability of different programming technologies used on different computational devices. Since the ACAF database is targeted to utilize GPUs, CPUs, FPGAs and a network, the used technologies include the following:

- OpenCL and pthreads for CPUs;
- OpenCL and CUDA for GPUs;
- OpenCL for FPGAs;
- MPI for a network.

Interoperability of the technologies mentioned above means the following functionality of the ACAF database: copying and/or converting of memory buffers from one technology into another; synchronizing the memory buffer content distributed between different technologies.

Basing on this information, we have extracted the important constructing blocks of the ACAF database design. These blocks are described in the following subsections. The full block diagram is shown at the end of this section in Figure 4.2.

4.2.4.1 Configuration

One of the input data a user should provide to the database is the configuration of the heterogeneous cluster utilization. The database is to be able to discover automatically the available and supportable hardware and technologies during the initialization phase. But only the user can define how to utilize the hardware. In particular, the user should specify:

- which network communication interface should be used, if any;
- which technology should be associated to one or another device;
- which amount of items should be distributed to the devices;
- some other miscellaneous device-dependent and technology-dependent parameters necessary for the execution.

The configuration is the same for all the running instances of the project in the cluster.

4.2.4.2 Context

The configuration defines the context for the database and the framework. The context consists of the device/technology pairs and the network interface. The device is a certain C++ object uniquely identifying a certain hardware device on the particular machine. The technology is an interface, which declares the necessary functions to execute the instructions on the supported devices. Only the supported devices can be coupled with a particular technology. The technology defines by itself the full set of the supported devices. Finally, the network interface declares the function set to perform network communication.

In contrast to the configuration, the context represents the actual set of devices available in the current system, as well as the actual device/technology coupling which is possible in the current framework version and setup.

4.2.4.3 Distribution

The context together with the configuration defines the distribution - a collection of the device/size and network node/size pairs. So, the distribution keeps information on the quantity of logical items to be stored on a certain device. Meanwhile, the network node sizes are automatically calculated and broad-casted over the predefined network interface. The correlation of the logical items count and the actual physical memory allocation is not a part of the distribution. The user can define several different distributions within the same configuration and use them for different aims.

4.2.4.4 Storage Objects

On the other hand, the context as well defines the storage objects - the instances of the storage interface. The storage interface declares the functional schema of operating with some physical memory space. Each storage object corresponds to some memory space (physical or virtual) and some programming interface for accessing this memory space. For example, the storage object can represent GPU memory space using OpenCL memory access functions, the main (RAM) memory space using the C++ memory functions or some remote network location accessible through the predefined network interface.

This means that the storage objects work with the low-level memory interactions. The storage objects do not know anything about the content of a particular memory block. The objects operate with byte-sized memory buffers.

4.2.4.5 Input and Output Data Definition

Another input a user provides is the definition of the input and output data of the algorithm. This definition describes the format of the data, the access, communication and synchronization schema, as well as the correspondence of the logical items count and the actual internal data items count.

4.2.4.6 Content Objects

Hence, the data description defines the content objects - the logic how to work with memory. In particular, a content object determines the following properties of the data:

- the type of the whole data collection;
- the types of the elements;
- the arrangement of the elements in the collection;
- the policy for reading the elements from a memory block;
- the policy for writing the elements;
- the possible directions and mechanisms of transferring the data;
- the correlation between the logical sizes defined by the distribution and the actual physical sizes of data.

4.2.4.7 Buffers

Finally, the content objects, the storage objects and a distribution define altogether the memory buffers. A memory buffer represents a range in certain memory space allocated by the storage object with the data arranged according to the content object. The size of the data is determined by the distribution with regard to the content factor. A memory buffer object contains only the reference to the memory region, the size of this region and the storage object, which manages this region.

4.2.5 Design of Framework

In order to implement and test the abilities of the proposed database design, it is necessary to develop the computational concept of the heterogeneous programming problem described in Section 4.2.3. The computational concept is the mathematical representation of an astrophysical simulation. This means that this concept can be described with an algorithm which evolves the state of the particle system and is able to operate on the data stored in the memory buffers of the database (see Subsection 4.2.4.7).

The mathematical background of an astrophysical simulation is a part of any astrophysical research, because this part represents the mathematical approximation of physical laws. The physical laws are a subject of the research:

- which laws are involved;
- which influence has a certain law, which of them are important and which of them can be ignored;
- how the laws work together;
- how a particular law should be approximated in order to be precise enough.

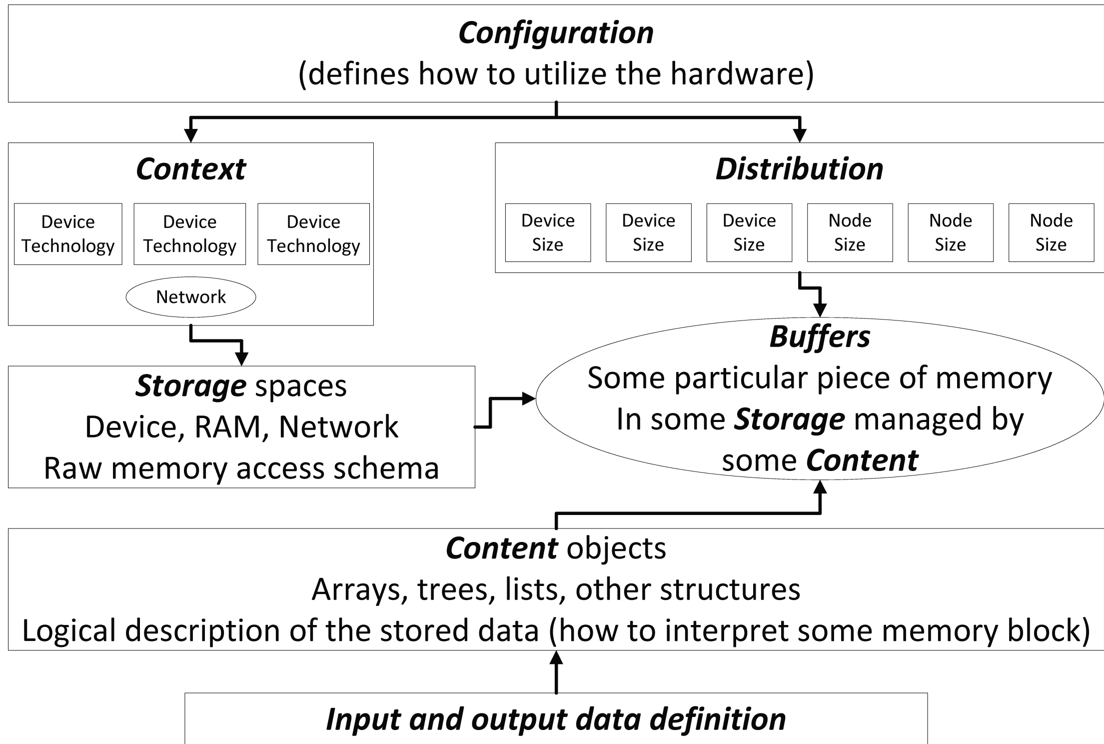

Figure 4.2: Diagram of the ACAF database design.

This means that the computational concept alone requires good programming skills from astrophysicists, because the precision of the simulation depends on the particular implementation of the mathematical algorithm. At the same time, the main difficulty in implementing the mathematical algorithm for a heterogeneous cluster lies in the necessity to implement the same algorithm several times for different technologies, which use different technology-dependent programming languages.

Consequently, it becomes reasonable to have some technology-independent programming language, which can be used for implementing the mathematical algorithm and can be afterward translated into the technology-dependent binaries. But as it was already mentioned in Section 4.2.3, in this work we concentrate on the database implementation. Therefore, the proposed language is left for the future work. Still in this section, we provide the architectural design of the whole framework, including the computational concept.

The design of the framework is schematically presented in Figure 4.3 together with the elements of the database design (see Section 4.2.4). Such design includes all the elements necessary to run an astrophysical simulation on a heterogeneous cluster.

4.2.5.1 Algorithm

Hence, for performing a simulation, the computational algorithm should be also provided by the user. The algorithm represents the mathematical approximations of the physical laws, which are aimed to evolve the state of the particle system.

4.2.5.2 Implementations

The computational algorithm together with the context object (see Subsection 4.2.4.2) defines a set of the technology-dependent and device-targeted implementations. Each implementation of this set represents a particular set of instructions, which can be executed on the target device. This means that each implementation is bound to the technology used in the current context for the device.

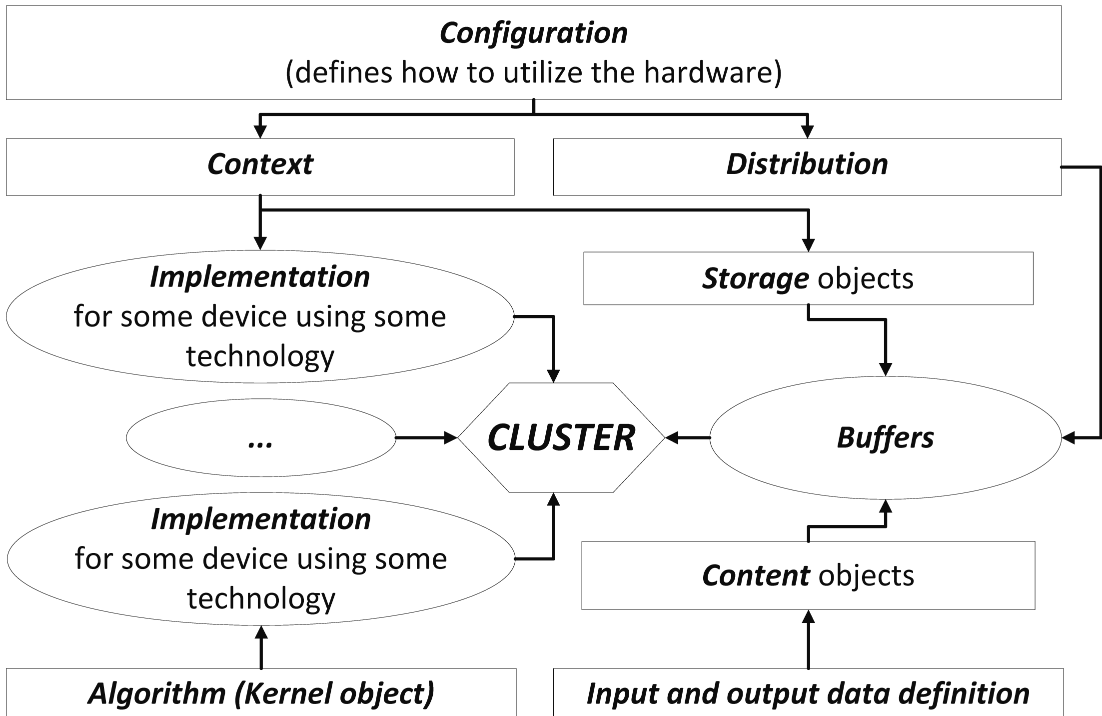

Figure 4.3: Diagram of the ACAF framework design.

4.3 Design of the Framework Implementation

In order to guarantee the correctness of the framework implementation and foresee the possible problems, we have firstly translated the proposed component-based framework design (see Figure 4.3) into the implementation design using the Unified Modeling Language (UML) diagram. The detailed description of the classes including some implementation details is provided in Section 4.4. This section gives the overview of the key mechanisms and techniques used in the implementation described in the following subsections.

4.3.1 Device Detection Mechanism

As described in Subsection 4.2.4.1 the first input data the framework expects from the user is the configuration. It provides the information how to utilize the hardware

presented in the cluster. But such a hardware-related specification can be quite complex due to the large variety of components presented in the cluster and different possibilities to use these components.

Therefore, in order to simplify and minimize the data necessary for the framework from a user, it was decided to implement the device detection mechanism. The idea of this mechanisms lies in detecting the available computational devices on each node and finding out which technologies can be used for programming these devices. Finally, the detection mechanism is to foresee some extending possibility for future devices and technologies.

Taking all these requirements into account, the mechanism is divided into 2 logical parts - a collection of independent *Architecture* subclasses and a collection of independent *Technology* subclasses. Each *Architecture* subclass and each *Technology* subclass has a descriptive unique string identifier available for the user (this identifier is not the C++ subclass name).

Each *Architecture* subclass represents a device type (CPU, GPU, FPGA and other) and provides an ability to enumerate all the devices in the current system of this type. The particular enumeration technique depends on the implementation of the subclass and the type of the device. In the current framework, there are 2 subclasses implemented:

- **CPUArchitecture** enumerates CPUs in the system. Since the current implementation targets Linux-based clusters, the CPUArchitecture class relies on the information provided in */proc/cpuinfo* file. The class parses the file on the initialization step and instantiates the *Device* objects.

- **GPUArchitecture** enumerates CPUs in the system. This class scans the whole Peripheral Component Interconnect (PCI) bus of the system in order to find the devices of the Video Graphics Array (VGA) type, which are in fact the GPUs. For each of these devices a *Device* object is instantiated.

Each *Technology* subclass represents a programming interface to interact with the devices. So, the framework requires that each subclass marks the devices supported by this interface. The marking process can be done in one of the following ways:

- The subclass checks the devices enumerated on the previous step by *Architecture* subclasses and for each device makes some tests in order to clarify the compatibility.

- The subclass scans the system for the available devices supported by this technology. (Usually, the interfaces provide the functions, which directly list the devices.) Afterwards the subclass matches the devices enumerated by *Architecture* subclasses and the devices listed by the technology.

The described device detection mechanism is a part of the framework initialization. This means that when the framework is successfully initialized, it has a list of device objects, where each object corresponds to a certain *Architecture* subclass and is supported by some *Technology* subclasses (none is also possible).

4.3.2 Configuration File

Having the device detection mechanism is not enough to make the correct decision how to utilize the devices of the cluster. Notwithstanding the fact that some heuristic-based decision is still possible, the user should be able to influence the utilization schema. Therefore, it is necessary to have a configuration file to specify the following parameters:

1. which of the supported technologies should be selected for the context for each device available in the system;
2. the fallback behavior in case of the unsuccessful association between devices and technologies;
3. the desired network interface, if any;
4. one or several distributions, where each distribution describes a partitioning of the logical units between the devices and the nodes of the cluster.

Taking into account the device information available after the framework initialization, it becomes possible to specify the device-technology association in the text form using the technology names, the architecture names and the full device names.

In order to simplify the usage and the implementation of the configuration file specification it was decided to use the libconfig[5] format for the file. The file has the following structure:

- there are 3 top-level sections: “context”, “network” and “distribution”;
- the section “network” is optional and specifies the network interface to be used, for example “MPI”;
- the section “context” is mandatory and includes device-technology associations (in the order of the processing priority):
    1. the technology name or the keyword “none” and an array of the full device names (as it was acquired by the *Architecture* subclass);
    2. the architecture name and the technology name or the keyword “none”;
    3. the optional parameter “skip”, which indicates if the unsuccessfully associated devices are to be skipped;
- the section “distribution” is mandatory and contains one or several named subsections;
- each named distribution subsection consists of device-specific blocks defining a certain partitioning of logical units:
    1. the full device name or the architecture name;
    2. the logical size and the block size vectors.

An example configuration file is demonstrated in Listing 5.1.

4.3.3 Context, Database and Distribution Initialization

Using the automatically listed devices set and the user-provided configuration file, the context object can be initialized. The context initialization is based on parsing the configuration file and traversing the device list, which was previously generated by the device detection mechanism. The result of the context initialization is a device-technology map. Only the devices listed in the map will be used later for the calculation. Another part of the context initialization is configuring the network interface according to the specification in the configuration file.

Using the initialized context, the database object can be initialized. The database initialization lies in the instantiation of the storage objects, responsible for device- and network-targeted transactions. The device-targeted storage objects are instantiated by the associated technology. The network-targeted storage objects are instantiated by the network interface. Each storage object is an instance of some *Storage* subclass (for different targets there are also other intermediate interfaces in the class hierarchy such as *DeviceStorage*, *LocalStorage*, *NetworkStorage*). Each storage object is fully responsible for providing the communication schema with its target. The result of the database initialization is a set of storage objects.

Finally, the initialized context and database make it possible to instantiate the distribution objects. A distribution object describes a certain partitioning of an astrophysical simulation problem between the different computational devices of the cluster. The partitioning is based on logical units. The distribution objects are composed using the initialized context and the user-provided configuration file. The initialized context lists the devices which will be used for the computation, while a particular subsection of the distribution section in the configuration file specifies the association of the devices to some size vector measured in the logical units. The logical units in the distribution object can represent either the actual particles count or some relative count, which can be converted to the particles count in the user-code using some factor.

The distribution initialization is divided into 2 steps:

1. at the first step, each network node initializes its local distribution for its own devices;

2. at the second step, the network nodes synchronize the sizes in order to gather the full distribution information.

4.3.4 Content Objects and Buffers Instantiation

As described in Subsection 4.2.4.5, the user also provides the information about the input and output data of the algorithm. This information is provided by instantiating the objects of some *Content* subclass. Each *Content* subclass provides a certain typical logical access schema. This schema includes the following characteristics:

- the data container type - for example one-dimensional array, multi-dimensional array, octree;

- the unit type - some scalar values (such as mass, temperature), some vector values (position, velocity, acceleration) or something else;

- the ownership of the data in the multi-storage context - which copy has the correct values for a certain range in case of the data being duplicated in several memory spaces;
- the synchronization mechanism in the multi-storage context - how the data should be transferred in case of the data being duplicated and kept up-to-date.

Another part of the content object instantiation is the memory allocation for storing the parts of data. Therefore, each content instantiation includes the units distribution object as a parameter. The units distribution object is generated from the regular distribution object using some factor. Having the units distribution, the content object is able to request the database to allocate the necessary amount of memory in the storage associated with the device. The result of this allocation is returned as an instance of a certain *Buffer* subclass. This instance includes internally the buffer physical size, the pointer to the managing storage object, the address of the actual buffer (an address form depends on the buffer type) and other storage-specific parameters.

4.3.5 The Computational Concept

As described in Section 4.2.5, the computational concept represents the mathematical algorithm of the particle system evaluation based on physical laws of the particles interaction. In the framework context, this mathematical algorithm is represented as a collection of *Kernel* class instances. Each instance is parametrized with a collection of technology-targeted implementations and execution parameters.

The technology-targeted implementations are created using the device-technology association available in the context and the technology-specific programming code. For the current implementation of the framework, the user should provide all the technology-specific programming code snippets for the technologies in use. The code will be compiled and prepared for each device associated with the technology, the resulting executable binary is represented by an object of some *Implementation* subclass. This object encapsulates all the parameters necessary to run the code on a certain device. All the device-targeted instances of *Implementation* subclasses for the particular mathematical computation are incorporated and managed by a single *Kernel* object.

The execution parameters can be either scalar values or content objects. The scalar values are byte-copied to the target device memory space. For each specified content object, the buffer allocated in the device memory is used.

4.3.6 Simulation Execution Principles

Taking all the described mechanisms into account, the user should perform the following steps for executing a simulation using the framework:

1. provide a configuration file using libconfig[5] syntax (see Section 4.3.2 for details and Section 5.1 for an example);
2. define which content objects are necessary for non-scalar distributed algorithm parameters (see Section 4.3.4 for details and Section 5.4 for an example);

3. write technology-specific execution code for all the technologies in use and bind them into the kernel objects (see Section 4.3.5 for details and Sections 5.2, 5.3, 5.4 for an example);

4. write the main execution logic using content objects and kernel objects in C++ language which usually consists of the following parts:

   - environment initialization;
   - objects construction;
   - data initialization;
   - time-evolving loop, which performs the kernel executions and content synchronizations.

   An example of the main execution logic can be found in Section 5.4.

The last step usually consists of a time evolving loop. For each iteration of this loop, some kernels are executed and some content objects are synchronized. Alternatively, some output data can be generated. But since the loop is written in C++ language it can include as many different instructions as necessary.

4.3.7 Considered Limitations

The proposed design considers the following limitations in the functionality and utilization of the framework:

- The design targets exclusively the data-parallel problems. Therefore, the implemented framework cannot be directly utilized for the task-parallel problems. This limitation is grounded from the properties of the astrophysical simulation problems described in Chapter 1. Moreover, the heterogeneous cluster computing is usually effective only for the data-parallel problems.

- The design and implementation of the framework targets x86-64 systems running a Linux Operating System. This limitation is formed by the statistics of TOP500 supercomputers, which shows that as of November 2015 90.6% of the supercomputers are x86-64 machines and 98.8% of the supercomputers run a Linux Operating System [6].

- The design of the framework maintains the separation of the computational resources by their type (*Architecture* subclasses) and the utilization schema (*Technology* subclasses). This separation forces a user to consider the specifics of the devices and to write the separate code for different devices in the current implementation. On the other hand, this limitation enables a user to finely-tune the computation code for each device according to the utilization schema and enhances the extendability of the framework in terms of the supported device types and utilization schemes.

4.4 Classes Description

This section provides the detailed description of all the key C++ classes implemented for the framework. For the interface classes, the implementation hierarchy is also described. All the implemented classes are organized in the C++ namespace *acaf* and the nested namespaces of *acaf*, where the nesting is reasonable for avoiding the name collisions. At any namespace level, a nested namespace with name *internal* is also used to express the implementation details, which are not supposed for the user code. The overview of the key classes and the dependencies between them is shown in Figure 4.4.

4.4.1 Basic Utility Classes

This subsection describes the set of the helper classes implemented to simplify and/or enhance the implementation of the classes hierarchy.

4.4.1.1 Handle and Class

The *Handle* class is an enhanced template smart-pointer. Most of the implemented classes inherit the *Class* class directly or indirectly. The most important part of the *Class* class is the protected reference counter, which is available for the *Handle* class and all the subclasses of the *Class* class. This reference counter together with the *Handle* class guarantees the robust smart-pointer implementation, which can be safely casted to a naked pointer and back to a smart-pointer without loosing the reference counting.

The template-based implementation of the smart-pointer also allows to use type-safe implicit conversions between the handles of different types, which lead to a safer "casts free" user code. Finally, the variadic template function *make_handle* voids the necessity to use naked memory allocations for the framework entities (e.g. *new* and *delete*).

4.4.1.2 Logger

The *Logger* class is a utility class in the current implementation of the framework, which has a set of static functions for logging some runtime information using a certain central place. The class provides an ability to log the information of different levels: errors, warnings, information, debug.

4.4.1.3 variant

The *variant* class is a helper class, which can hold an object of any type without being a template class. An object of the *variant* class can be easily forwarded to any functions as an argument and cloned as necessary. The *variant* class helps to avoid the template-based implementations of the type-dependent classes and functions, where the framework structures deal as a proxy between the sender user function and the receiver user function maintaining the type and memory safety. This means that if the actual type of some parameters and arguments does not matter to the internal framework functionality, but is to be safely forwarded to some callback user functions,

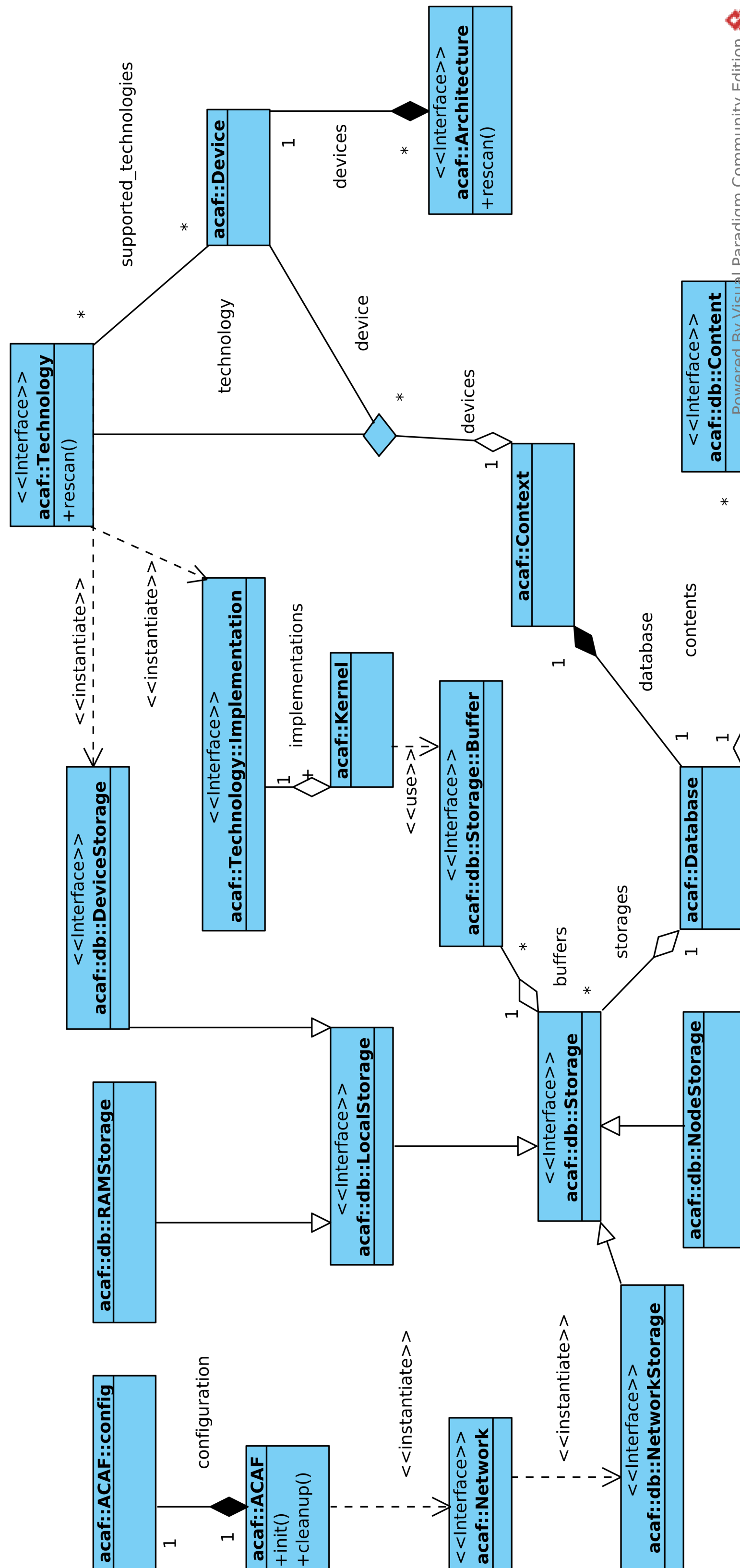

Figure 4.4: The overview UML diagram of key classes of ACAF.

the *variant* class becomes essential. The class idea and implementation are inspired by the *boost::any* class.

4.4.1.4 vector_t

The *vector_t* class is a template class, which aims to encapsulate vector arithmetic operations and provide an ability to work with constant-length vector variables as with scalar variables. Internally, the class stores the data in the linear piece of memory. The size of the memory piece is equal to the length of the vector multiplied by the size of each item. The number of items in the vector should be a power of 2 and not greater than 32. The class contains the implementation of the following operations:

- for the arithmetic operations with other vector variables and scalar variables;
- conversion between different vector template specializations;
- operators to access the individual items;
- more complex vector-specific functions such as calculating the eucleadean length, the volume of the vector, the dot of 2 vectors, the cross of 2 vectors.

4.4.1.5 ErrorCode

The *ErrorCode* class is a service class, which provides an ability to track and handle the error codes in the application. The class itself represents a wrapper for an integer variable, which is the actual error code. The meaningful description of the error code is stored internally in the helper *StoredErrorCode* class. The instance of this class is unique for each integer error code and each string error name. The framework user and the framework developer can instantiate the *ErrorCode* class objects using either an integer code or the error name. The latter option either will result in the existing integer code or a new negative integer code will be registered and associated with the provided error name. There are 2 reserved pre-instantiated error codes:

- the error name *Success* is associated with the error code 0;
- the error name *Unknown* is associated with the error code -1.

Another possibility to define some error code is provided by two preprocessor macros: *ACAF_DECLARE_ERROR* and *ACAF_DEFINE_ERROR*.

4.4.1.6 Option

The *Option* class is a service class, which is used to manage the available command line options. The class declares the semantics of a command line option, using the static storage structures to organize the options. The framework predefines several command line options. Each part of the framework and the user code can define additional options. The definition of the custom command line options can be done using the declared C++ macros or calling manually *Options::addOption* function with the appropriate arguments. The sole requirement to the usage of the macros and the

function is that all options should be added to the framework before the *ACAF::init* call. The macros are designed to add the options in the static manner.

4.4.2 ACAF

The *ACAF* class is an entry point to the whole framework functionality. The class has the following public methods:

- *init* - completely initializes the framework: parses the command-line arguments, parses the configuration file, scans the system for available devices, initializes the basic entities: context, database;
- *cleanup* - destroys all resources allocated by the framework since the initialization;
- *conf* - returns the parsed configuration tree as a pointer to the root *ACAF::config* object.

Parsing of the command line arguments is based on the statically assembled collection of possible options (a helper class *Option* is used for it). The parsing of the configuration file is done firstly by libconfig[5] library, which checks the syntax of the input file and constructs the corresponding object-based tree. Afterwards, *ACAF* class converts the constructed object-based tree into the *ACAF::config* objects.

ACAF::config class represents a node of the configuration file and has methods to list the children nodes, select a particular child node by name and retrieve the value of the node.

4.4.3 Device

The *Device* class is one of the central classes in the framework implementation. An instance of this class represents a device in the current system. The class has the following private member fields:

- the vendor name as a string field and the vendor identifier as a variant architecture-dependent field;
- the device name as a string field and the device identifier as a variant architecture-dependent field;
- the pointer to the instance of the *Architecture* subclass, which has created this device object;
- the map of supported technologies and technology-specific identifiers of the device;
- the set of some architecture-defined, technology-defined or custom device properties.

Functionally, the *Device* class is simple and does not perform any tasks. All modification operations of the device are not public and can be called only by the friend classes. A single exception is adding of custom properties.

4.4.4 Architecture

The *Architecture* class is a common interface for all different computational device architectures. Under the device architecture we understand the design architecture of the device processing unit, which can be used for the computational purposes (e.g. CPU, GPU, FPGA). The interface declares the common member functions and member fields for all the architecture subclasses. The main function of any *Architecture* subclass lies in enumerating the devices of some specific type. The framework relies on the unambiguous correspondence of the devices and the supported architectures: there is no such device which can belong to more than one architecture.

Additionally, the *Architecture* class defines the static mechanism guarantying that each subclass is instantiated only once in the scope of one running process. This mechanism is based on the statical singleton instantiation of the *Architecture* subclasses during the framework library loading. All instances are stored in the static name-object map. Only the instances in the map will be taken into account by the framework.

To support some other computational architectures as the predefined ones, the user should implement another subclass of the *Architecture* class. The new subclass should provide the framework with the actual implementations of 2 pure virtual methods of the interface:

- *getName* - returns the name of the architecture;
- *rescan* - rescans the entire system in order to detect all available devices of the current architecture type; and stores the appropriate *Device* instances in the member variable.

The current framework implementation includes 2 subclasses of the *Architecture* class: *CPUArchitecture* and *GPUArchitecture* (see Figure 4.5).

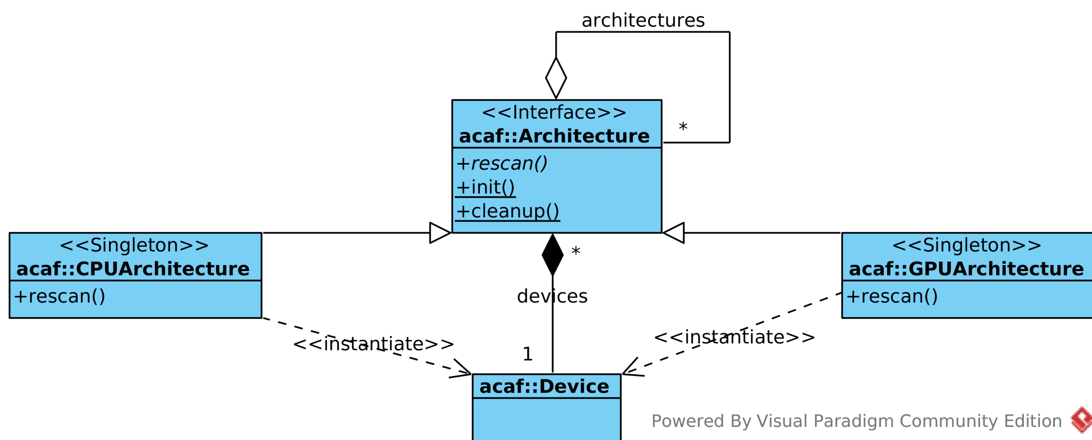

Figure 4.5: The UML diagram of Architecture subclasses.

4.4.4.1 CPUArchitecture

CPUArchitecture subclass is implemented to list CPUs available in the system. The subclass logic is based on the Linux-like representation of CPUs enumerating, particularly the subclass parses */proc/cpuinfo* file, which is a standard Linux kernel interface to represent the CPU information. During the parsing of the file, the *CPUArchitecture* subclass collects information relevant to the calculation:

- the number of processors;
- the number of cores for each processor;
- the physical location of each processor;
- the vendor identifier of each processor;
- the model name of each processor.

The *CPUArchitecture* subclass creates the *Device* instances for each physical processor independent of the number of cores it has. The architecture-dependent device identifier is the physical slot number of the processor.

4.4.4.2 GPUArchitecture

GPUArchitecture subclass is implemented to list GPUs available in the system. The subclass logic is designed for PCI-connected devices, since the subclass scans the PCI bus of the system and lists all the devices of the VGA type and creates for each of them a *Device* instance. The new instance stores the following properties of the found PCI device:

- the PCI-based vendor name and vendor identifier;
- the PCI-based device name;
- the PCI bus address of the device - the quad of domain, bus, device and function variables, which is also the architecture-dependent device identifier.

4.4.5 Technology

The *Technology* class is a common interface for all computational technologies. The interface declares the common member functions and member fields for all the computational technology subclasses. Any *Technology* subclass implements the following interface functions:

- *getName* - returns the name of the technology;
- *rescan* - scans the entire system to identify the devices supported by the technology and matches the devices which were previously listed by some *Architecture* subclass. The matched instances are marked as supported with the technology. The particular matching mechanism depends on the technology type and the implementation: some technology can enumerate the devices directly, the other check the devices listed by the architectures to fulfil some criteria;

- *getStorage* - for each supported device the subclass should provide an instance of some *Storage* subclass, which is able to manage the device memory space;
- *implement* - for each supported device the subclass should provide an instance of some *Technology::Implementation* subclass, which is able to execute some programming code or some binary on the device. Usually, each *Technology* subclass provides also an implementation of the appropriate *Technology::Implementation* subclass.

The *Technology* class guarantees the singleton instantiation of the subclasses during the framework loading in the same way as the *Architecture* class.

Extending of the supported computational technologies can be done by implementing another subclass of the *Technology* class. The new subclass should provide the framework with the actual implementations of the 4 pure virtual methods mentioned above.

The current framework implementation includes 3 subclasses of the *Technology* class: *PthreadTechnology*, *OpenCLTechnology* and *CUDATechnology* (see Figure 4.6).

4.4.5.1 PthreadTechnology

PthreadTechnology subclass is implemented to execute the computational code on CPUs in multiple threads using *pthreads* library. In addition to pthreads library, the subclass uses hwloc library in order to bind the threads to CPU cores, where it is possible. This technology is supported by any device instantiated with *CPUArchitecture* subclass. This means that the *rescan* method lists all the *CPUArchitecture* devices. The *RAMStorage* class is used for the devices within this technology. The *implement* method results in the instance of the inner class *PthreadTechnology::Implementation.*

For running some computations on the devices, the class creates for each CPU core a single thread, which runs an idle function waiting continuously for a kernel. The technology uses semaphore, mutex and condition synchronization models in order to ensure that the threads remain idle and do not consume any CPU time until a new kernel is queued.

PthreadTechnology::Implementation subclass represents a collection of jobs to be executed within the threads created by the technology for a particular device. This representation includes the following parameters:

- the pointer to the actual computational function, which is also the sole argument in the variadic list for the *implement* method of the *PthreadTechnology* subclass.
- the number of tasks to be executed within the kernel: how many times the actual computational function should be called for different data in order to consider the computation completed.
- the execution statuses for each task: the return value of the actual computational function has the *acaf::status* type and it can be evaluated to some error code.

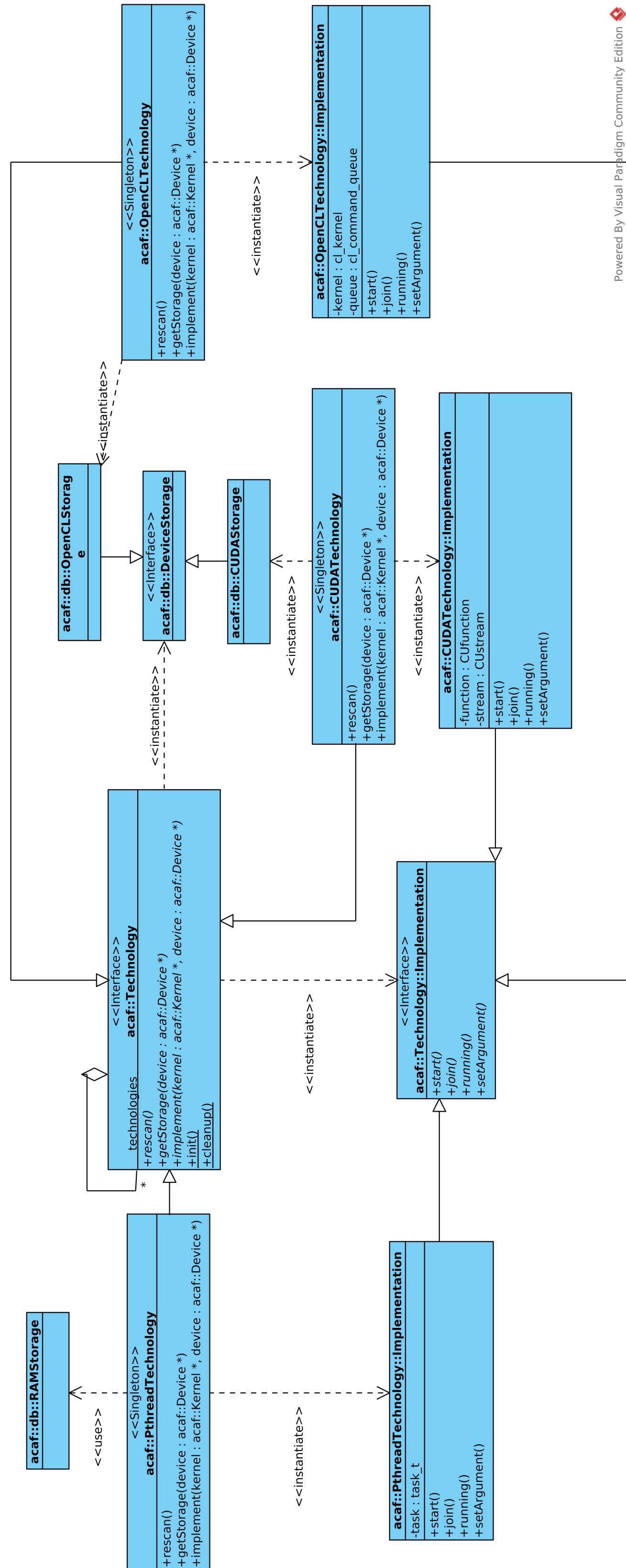

Figure 4.6: The UML diagram of the Technology subclasses.

4.4.5.2 OpenCLTechnology

OpenCLTechnology subclass is implemented to execute the computational code on any supported device using OpenCL library. The supported devices are identified in *rescan* method by querying OpenCL library for all the devices available in the system; and matching these devices with the devices enumerated previously by the *Architecture* subclasses. In particular, the current implementation of the framework includes the logic to match the following devices:

- the devices of the CL_DEVICE_TYPE_CPU type are supposed to be listed by the *CPUArchitecture* subclass. Therefore, the technology searches by the device name and the vendor name among the devices of this architecture;
- the devices of the CL_DEVICE_TYPE_GPU type are supposed to be listed by the *GPUArchitecture* subclass. Therefore the technology tries firstly to query OpenCL for the PCI bus address and match it to the known *GPUArchitecture* devices (using the OpenCL extensions of AMD and NVIDIA GPUs). If matching PCI bus ID appeared impossible or unsuccessful, then matching by the vendor name and the device name is used;
- the devices of the CL_DEVICE_TYPE_OTHER type are currently ignored by the technology, but generally can be also matched to some listed devices using the same mechanisms as for the previous types.

For running some OpenCL code on the devices, an instance of the inner class *OpenCLTechnology::Implementation* is instantiated. The data management and transferring is implemented in the *OpenCLStorage* class.

OpenCLTechnology::Implementation subclass represents a wrapper for the OpenCL kernel structure. This subclass can be used for executing some code on OpenCL-enabled devices. This subclass is instantiated by the *implement* method of the *OpenCLTechnology* subclass. The instantiation is possible using either a string variable with the program written in the OpenCL C language or a file name containing such a program. *OpenCLTechnology* subclass compiles the program code into an OpenCL kernel object. The latter together with the target device comprises the parameters of the *OpenCLTechnology::Implementation* subclass. Another possible argument is the compilation options for the OpenCL implementation.

4.4.5.3 CUDATechnology

CUDATechnology subclass is implemented to execute the computational code on any CUDA-designed device. The technology uses CUDA Driver API in order to match devices, control the data flow and the execution of the binary code on devices. The usage of CUDA Driver API allows to make the framework more independent:

- a program based on CUDA Driver API requires only the CUDA library delivered together with the graphics device driver on the target system;

- the CUDA Driver API can utilize the binary Parallel Thread Execution (PTX) modules to execute some functions on GPU;
- the CUDA Driver API supports for multiple GPUs is based on context CUDA objects; this simplifies the management of the devices at the framework level.

Matching CUDA-enabled devices to the devices enumerated previously by the *Architecture* subclasses relies on the fact that the current CUDA-enabled devices are connected over the PCI interface. This means that any device can be identified by its PCI bus address. This address is used in order to find the matching framework devices. The *implement* method creates an instance of the inner *CUDATechnology::Implementation* class. The data communication mechanisms are provided by the *CUDAStorage* class.

CUDATechnology::Implementation subclass represents a wrapper for the CUDA binary function, which can be used for executing some code on CUDA-enabled devices. This subclass is instantiated in the *implement* method of the CUDATechnology subclass. The instantiation is possible using either a string variable with the PTX binary code or a file name containing such a cubin, PTX or fatbin binary code, which are the possible arguments in the variadic list. To produce PTX, cubin, fatbin binary codes the user should compile the original CUDA C program using the nvcc compiler, which is a part of the CUDA Software Development Kit (SDK) package. Still, Parallel Thread Execution (PTX) is a pseudo-assembly language and therefore it can also be used to write a CUDA-targeted code manually. Using the provided PTX binary code and a function name, the *CUDATechnology* subclass creates a CUfunction object.

4.4.6 Network

The *Network* class is a common interface for different network interfaces. The interface declares the common member functions and member fields for all the network subclasses. The main function of any *Network* subclass lies in defining the communication between the different nodes of the network. The *Network* object is a singleton for each running instance of the program. This means that only one instance of a particular *Network* subclass can exist in the scope of a single running process.

To support some other network interfaces as the predefined ones, the user should implement another subclass of the *Network* class. The new subclass should provide the framework with the actual implementations of the 10 pure virtual methods of the interface:

- *getNodesCount* -returns the total number of the nodes in the network;
- *getMyNodeIdx* - returns the current node identifier;
- *allgather* - gathers the whole buffer from all the nodes;
- *alltoall* - gathers the whole buffer from all the nodes and redistributes it again between all the nodes;
- *send_bcast* - sends a broadcasting message to all the nodes in the same network/-subnetwork as the current node;

- *recv_bcast* - receives a broadcasting message from the node;
- *send* - sends the content of some buffer to the remote buffer;
- *recv* - receives the content of the remote buffer in some local buffer;
- *init* - initializes the instance of the class;
- *instantiate* - for each node in the network creates an instance of some *Network-Storage* subclass and adds it to the database.

The current framework implementation includes 1 subclass of the *Network* class: *MPINetwork*.

4.4.6.1 MPINetwork

MPINetwork subclass provides an implementation of the *Network* class using the Message Passing Interface (MPI) library. The MPI library is widely used in the heterogeneous applications and can fully reflect the whole set of the *Network* interface functions. The current implementation of *MPINetwork* subclass works only with the global MPI_COMM_WORLD communicator and does not support grouping of the nodes. The implementation of the *instantiate* function creates for each node in the global communicator an instance of the *MPIStorage* class.

4.4.7 Context

The *Context* class represents a map of device-technology pairs. Each pair describes the utilization schema of the device presented in the system. The context is a singleton object for each running instance of the program. The context is initialized using the global ACAF::config based configuration. The *Context* class also hosts an instance of the *Database* class. When the initialization of the context is finished, the member database will be also initialized.

4.4.8 Storage

The *Storage* class is the interface for all the framework entities which represent the engines for writing and reading the data to/from some memory space. All actual storage entities should inherit this class directly or indirectly in order to be correctly processed by the other framework entities. The interface class contains the declarations of the basic functions and also the trivial implementations for some of them. The interface class holds a collection of all owned memory buffers as a map of buffer-content pairs. All created buffers represent some pieces of memory with no connection to the particular format of the stored data (*Content* class). The most important methods of the class include:

- *init* - initializes the current instance of the class. The basic implementation adds the current instance to the database set of storage objects. All subclasses should call the parent init function in order to make sure that all the parts of the class are correctly initialized.

- *create* - creates a new buffer object for the specified content object and the necessary physical buffer size. Every *Storage* subclass creates an instance of some particular buffer class, which fits the aims and the functionality of the class.
- *find* - for the specified content finds the buffers owned by this storage object.
- *isSame* - checks if the current instance represents the same memory space as the instance passed over the arguments.

Usually, the actual storage objects do not inherit the *Storage* class directly, but inherit special subclasses, designed to simplify the implementation. Still, the user is able to extend the framework according to the research needs and implement the new storage types inheriting either the *Storage* class itself or some of its subclasses.

Storage::Buffer is an inner class of the *Storage* class. It declares the main interface for the buffer objects. A buffer object is a wrapper for some region in some memory space. The *Storage::Buffer* class declares the general functions for all the buffer subclasses. it is supposed that the actual memory space wrapped with the particular buffer object is used only by the "parent" storage class and its subclasses. Therefore, the *Storage::Buffer* class does not declare any memory access functions. The interface has one pure virtual function - *isSame*, which checks if the current buffer object wraps the same memory region as the object passed over the arguments.

4.4.8.1 LocalStorage and NetworkStorage

The first level in the hierarchy of the different *Storage* subclasses represents the classification of the storage objects according to the memory space location in the system:

- the *LocalStorage* class represents an interface for any memory space, which is physically located within the current network node. The interface includes the functions for the basic data manipulation: read, write, fill, map, unmap.
- the *NetworkStorage* class represents an interface for the remote-located memory space. The policy of the framework declares that each network node corresponds to an instance of some *NetworkStorage* subclass. The basic data manipulation for the remotely located memory space is not effective. Therefore the interface does not provide any functions similar to the *LocalStorage* class. The data manipulation in this case should be managed over the *Network* class.

4.4.8.2 DeviceStorage

The *DeviceStorage* class is a subclass of the *LocalStorage* class and represents some memory space located on a particular device (usually an accelerator) attached directly to the current node. In addition to the functions and variables inherited from the superclasses, the *DeviceStorage* class stores a pointer to the target *Device* instance and checks its validity when necessary. Also, the class contains an inner class *DeviceStorage::Buffer*, which does not provide additional functionality, but addresses the classification of buffers at the type level.

4.4.8.3 OpenCLStorage

The *OpenCLStorage* class is a subclass of the *DeviceStorage* class and represents the memory space used for some OpenCL device. In the initialization function of the storage class, the device is checked to support OpenCL technology and the OpenCL device identifier is acquired. The storage class creates a separate OpenCL command queue for data transferring. The class also provides its own buffer inner class *OpenCLStorage::Buffer*, which wraps an object of *cl_mem* type. The storage class uses OpenCL functions to create the buffer and transfer the data to/from the device buffers.

4.4.8.4 CUDAStorage

The *CUDAStorage* class is a subclass of the *DeviceStorage* class and represents the memory space used for some CUDA device. In the initialization function of the storage class, the device is checked to support CUDA technology and the CUDA device identifier is acquired. The storage class creates a separate CUDA processing stream for data transferring. The class also provides its own buffer inner class *CUDAStorage::Buffer*, which wraps an object of *CUdeviceptr* type. The storage class uses CUDA Driver API functions to create the buffer and transfer the data to/from the device buffers.

4.4.8.5 RAMStorage

The *RAMStorage* class is a subclass of the *LocalStorage* class and represents the RAM memory space of the current node. The *RAMStorage* object is a singleton for each *Database* object. This means that there is one and only one object of the *RAMStorage* type in the storage collection of each *Database* object. The class uses the raw memory access functions for data operations and malloc, free functions for allocating and freeing the memory buffers. The class also provides its own buffer inner class *RAMStorage::Buffer*, which wraps a raw memory pointer (void *).

4.4.8.6 MPIStorage

The *MPIStorage* class is a subclass of the *NetworkStorage* class and represents the remote memory accessed with MPI functions. As it was mentioned in Subsection 4.4.8.1, the *MPIStorage* class does not provide any functions to transfer the data, but enables the mirroring of the remote buffers in the local database. So, the class has the functions to create a buffer object and an implementation of the buffer inner class *MPIStorage::Buffer*, which wraps a remote memory buffer identified by the MPI tag. The MPI tag of the buffer is used by the *MPINetwork* class in order to mark the messages between the nodes, which facilitates the correctness of the network transfers. The MPI tags are assigned per content and the framework relies on the same content instantiation order of all the content objects.

4.4.8.7 NodeStorage

The *NodeStorage* class is a special class used to acquire temporary memory buffers for synchronization purposes (it extends the *Storage* class directly). The class operates also

with the main RAM memory space, as well as the *RAMStorage* class. But in contrast to the *RAMStorage* class, the *NodeStorage* class does not provide an ability to create a usual buffer. It only creates a temporary buffer, which is a raw access buffer and can be used for data operations.

4.4.9 Distribution

The *Distribution* class addresses the problem of defining the data separation between the network nodes and between the devices inside of each node. For these purposes an instance of the class stores 2 data maps: the device-partition pairs map and the network node-partition pairs map. An instance of the *Distribution* class can be created using some named distribution section of the configuration file and a *Context* object. The *Context* object defines the devices and the network nodes which should be included in the distribution maps. At the same time, the named section of the configuration file defines the partition sizes for each element of the maps.

The distribution section in the configuration file consists of blocks. Each block refers either to some particular device by its name or to all the devices of some architecture. The blocks referring to the devices by name have a higher priority. The *Distribution* class checks each block of the section and scans the context for the mentioned devices. For each found device, the class assigns the partition size in the map. In the current implementation of the framework only the pure numbers are allowed as the partition sizes. Still, the simple math operations relative to the device properties (such as the number of compute units, preferred work group size) will simplify the format of the configuration file.

When the device map is complete, the distribution class calculates the total partition size of the current node and synchronizes the partition sizes between all the nodes in the network filling out the corresponding map. To traverse the defined distribution, the class provides the iterator implementation and search function.

In addition to the devices partitioning map and the network partitioning map, the *Distribution* class contains a device-block sizes map, which is also initialized with the named distribution section of the configuration file and addresses the processing granularity. The usage of the block size parameter depends on the selected technology (for example, it defines the work group size for OpenCL and the items count per job for pthreads).

The *Distribution* class has the sizes to be defined in some abstract common units (usually, particles count). At the same time, the content object creation function needs a distribution of its items, which does not often respect one-to-one correspondence to the abstract units (1 content item does not always correspond to 1 particle). Therefore, the framework defines another interface class *UnitsDistribution*, which should be used for the content object creation. The *Distribution* class provides a function *units*, which simplifies the conversion of the *Distribution* instance into the *UnitsDistribution* instance for the usual case of the factor correspondence between the content items count and the distribution units (for example, 1 particle corresponds to a constant number of items).

4.4.10 Content

The *Content* class is a base interface for all the classes which represent the data layout for some physical memory block. Each final implementation of the *Content* interface stores the full set of the buffers. This corresponds to the whole data range processed in the application. The interface declares some common functions for all the content objects:

- *fill* - fills the whole data range with some constant value;
- *random* - fills the whole data range with some random values;
- *synchronize* - synchronizes the content of the buffers in the current node with the other network nodes;
- *isSame* - checks if the current content object represents the same data as the one passed over the function arguments.

Each *Content* object is bound to some *Context* instance, since it defines which devices and network nodes are taken into account. it is supposed that *Content* objects should only be instantiated by some *Database* instance. But the user is able to implement any other classes which correspond to the *Content* class concept extending the possibilities of the framework. The current framework implementation provides 2 contents: a local array and a synced array.

4.4.10.1 LocalArray

The *LocalArray* class is a subclass of the *Content* class and represents a linear items layout in memory. The items are distributed over the network nodes and the devices, but without any synchronization schema. This means that the data is partitioned according to the supplied *UnitsDistribution* instance. But each part exists independently to the other parts. Still, the object stores the full distribution map of storage-partition pairs.

4.4.10.2 SyncedArray

The *SyncedArray* class is a subclass of the *LocalArray* class and represents a linear items layout in memory. The items are distributed over the network nodes and the devices with the full range available in each part of the system and synchronization possibility. This means that the data is partitioned according to the supplied *UnitsDistribution* instance. But each buffer keeps the full range of the array stored. So any computational device can access the items values stored in the other buffers. By the user request, the data in the buffers can be synchronized. The synchronization of the data is performed in the following steps:

- the data from the local storage buffers is collected into a temporary buffer acquired from the *NodeStorage* instance;

- the temporary buffers of different network nodes are synchronized using the *all-gather* function of the *Network* instance;
- the synchronized data is written back from the temporary buffer to all the local storage buffers.

4.4.11 Database

The *Database* class represents the central storage for all the data-related objects in the framework environment. Primary the database holds the following objects:

- a set of the *Storage* objects, where each object refers to some memory space (see Subsection 4.4.8);
- a map of the named *Content* objects, where each object represents some data layout schema used in the user application (see Subsection 4.4.10).

If the storage objects are created by some other entities of the framework and just added to the appropriate *Database* instance, the *Content* objects are directly instantiated by the *Database* instance. For this purposes, the class defines a template function *create*, which takes a particular subclass of the *Content* class as a template argument. Also, the *create* function needs a name for the content and an instance of the *UnitsDistribution* class to initialize the newly created content object and to add it to the named map.

4.4.12 Kernel

The *Kernel* class represents a function running distributively with all its implementations for the available devices and all the necessary function arguments. The *Kernel* class is an end-user class. This means that the framework user is able to instantiate as many objects as necessary. The class provides the following manipulation functions:

- *add* - creates and adds an implementation of the kernel. The actual creation of the implementation object is forwarded further to the specified technology instance. The creation of the implementation object is done separately for each device assigned to the specified technology. Only the devices in the context of the kernel are taken into account.
- *set* - adds a value or a *Content* object to the list of the arguments of the kernel. A scalar argument value is passed to each implementation as it is. The *Content* object is passed to each implementation in the form of the buffer corresponding to the device, where the implementation is executed.
- *start* - triggers the concurrent execution of all the available implementations of the kernel.

The order of the described functions reflects the usual work flow with a kernel object:

1. The user creates a named kernel object.

2. The user adds several implementations of the kernel for different technologies.
3. The user sets the necessary execution parameters of the kernel.
4. The user starts the kernel execution.

4.4.12.1 Technology::Implementation

The inner *Technology::Implementation* class is an interface base class for all the kernel implementations. The interface class has only pure virtual functions, which define the possible operations available for the kernel implementation:

- *set* - sets the provided value or the content buffer to the implementation according to the technology;
- *start* - starts the execution of the implementation;
- *running* - returns true if the implementation is currently running and false otherwise;
- *join* - blocks the further application running till the implementation execution is finished.

Chapter 5

Results

5.1 The Usage Example

The results of our research are represented with the implementation of the framework with the proposed design. To present the abilities of the framework and to evaluate the final implementation, a usage example was developed. The usage example solves a typical astrophysical force N-Body simulation problem described earlier in Section 1.2. The usage example performs the described simulation on the heterogeneous cluster providing the possibility to test different configurations: CPUs only, GPUs only, CPUs and GPUs simultaneously, using a single node or the whole cluster. The initial state of the particle system is described by random particle positions, equal particle masses and zero particle velocities.

The usage example consists of 4 distinct parts: the configuration file for the framework; the kernel implementations for pthread and OpenCL technologies; the main function using framework API. All these parts are described in details in the following subsections.

5.1.1 The Configuration File

The example configuration file is presented in Listing 5.1.

Listing 5.1: "Configuration Example"

```
1 network="MPI";
2 context: { skip = true; CPU = "pthread"; GPU = "OpenCL"; };
3 distribution: {
4   default = (
5     { architecture = "GPU"; size = [1024]; block = [256]; },
6     { architecture = "CPU"; size = [256]; block = [4]; }
7   );
8 };
```

As described in Section 4.2.4.1, the example configuration file provides the framework with the hardware utilization schema. In particular, line-by-line the example file defines the following:

1. The "network" parameter in line 1 defines that MPI should be used for the network communication within the cluster network. The calculation will be distributed over all the active nodes of the cluster. Eliminating this parameter will lead to the single-node computation.

2. The "context" parameter in line 2 provides the textual context definition. All the CPU devices will be utilized by the pthread technology; all the GPU devices will be utilized by the OpenCL technology; all the other devices or the devices of the previous type not supported by these technologies will be skipped without producing any errors. Changing the "skip" parameter to value "false" will lead to the errors if there are any CPUs or GPUs not-supported by the assigned technologies.

3. The "distribution" section in lines 3-8 defines one entity with the name "default", which prescribes the following partitioning of the problem:

   (a) Each GPU device processes 1024 items per iteration, calculating 256 items per work group (line 5).

   (b) Each CPU device processes 256 items per iteration, calculating 4 items per thread job (line 6).

5.1.2 OpenCL Kernel Implementation

The example OpenCL kernel implementation is provided in Listing 5.2.

Listing 5.2: "OpenCL Algorithm Example"

```
1 #pragma OPENCL EXTENSION cl_khr_fp64 : enable
2 #pragma OPENCL EXTENSION cl_amd_fp64 : enable
3
4 #define SOFTENING 0.001
5
6 __kernel void force ( uint4 acaf_total ,
7   __global double * mass, __global double4 * position ,
8   __global double4 * velocity , double time_step )
9 {
10   __local double4 shared_position[ITEMS_PER_GROUP];
11   size_t lid = get_local_id(0);
12
13   shared_position[lid] = position[get_global_id(0)];
14   double4 this_acc;
15   this_acc.x = this_acc.y = this_acc.z = this_acc.w = .0;
16   for ( size_t i = 0; i < acaf_total.x; ++i )
17   {
18     double4 dist = shared_position[lid] - position[i];
19     this_acc += mass[i] * dist / powr(length(dist) + SOFTENING
         , 3.);
20   }
```

```
21
22   size_t gid = get_global_id(0);
23   velocity[gid] += this_acc * time_step;
24   position[gid] += velocity[gid] * time_step;
25 }
```

This kernel implementation provides the computational code for evolving the position of the single particle written in OpenCL C language. The first argument of the kernel function represents the special framework argument, which hints how many particles are represented in the full data set (in other words, what is the size of the arrays “mass”, “position”, “velocity”). The index of the current particle is provided by OpenCL C *get_global_id* function.

5.1.3 pthread Kernel Implementation

The example pthread kernel implementation is provided in Listing 5.3.

Listing 5.3: ”Pthread Algorithm Example”

```
1 #define SOFTENING 0.001
2
3 status force (
4   const acaf::uint4 & jid, const acaf::uint4 & jtotal,
5   const acaf::variant_vector & args
6 )
7 {
8   double * mass = reinterpret_cast<double *>(*(args[0].get<
      void *>()));
9   double4 * position = reinterpret_cast<double4 *>(*(args[1].
      get<void *>()));
10  double4 * velocity = reinterpret_cast<double4 *>(*(args[2].
      get<void *>()));
11  double time_step = *(args[3].get<double>());
12
13  double4 this_pos = position[jid[0]];
14  double4 this_acc (0.);
15  for (size_t i = 0; i < jtotal[0]; ++i)
16  {
17    double4 dist = this_pos - position[i];
18    this_acc += mass[i] * dist / pow(dist.length() + SOFTENING
        , 3.);
19  }
20
21  velocity[jid[0]] += this_acc * time_step;
22  position[jid[0]] += velocity[jid[0]] * time_step;
23
24  return error::Success;
25 }
```

This kernel implementation provides the computational code for evolving the position of the single particle written in C++ language using the special framework types. The pthread kernel implementation function arguments are strongly prescribed to be the following:

1. the first argument is a uint4 vector, which delivers the identifier of the current particle;
2. the second argument is a uint4 vector, which delivers the number of all the particles in the full data set;
3. the last argument is a vector of variants, which contains the user arguments. The arguments should be properly casted before usage.

5.1.4 The Main Function

The example main function of the program with the appropriate calls of the framework API is provided in Listing 5.4.

Listing 5.4: "Main Function Example"

```
1 int main(int argc, char ** argv)
2 {
3   MPI_Init(&argc, &argv);
4   status s = error::Success;
5   do
6   {
7     s = acaf::initialize(argc, argv);
8     if (s.fail()) break;
9
10    Handle < DataBase > db = Context::getContext()->getDB();
11    LinearParticles distr(Context::getContext(), acaf_string("
          default"));
12    Handle<Content> mass, pos, velo;
13
14    {
15      acaf::pair<Handle<Content>, status> tmp;
16      tmp = db->create< SyncedArray<double, 1> >("mass", distr
            .units(1));
17      if (tmp.second.fail()) cout << tmp.second;
18      mass = tmp.first;
19      mass->fill(acaf::variant(1.));
20      tmp = db->create< SyncedArray<double4, 1> >("position",
            distr.units(1));
21      if (tmp.second.fail()) cout << tmp.second;
22      pos = tmp.first;
23      pos->random(
24        acaf::variant(double4({-1., -1., -1., 0.})),
```

```
25          acaf::variant(double4({2., 2., 2., 0.}))
26        );
27        tmp = db->create< LocalArray<double4, 1> >("velocity",
             distr.units(1));
28        if (tmp.second.fail()) cout << tmp.second;
29        velo = tmp.first;
30        velo->fill(acaf::variant(double4(0.)));
31      }
32
33      double current_time = 0.;
34      double end_time = 1.;
35      double time_step = 0.01;
36
37      Kernel force("force", Context::getContext());
38      s = force.add("OpenCL", "gravity.cl", "-cl-mad-enable",
           true);
39      if (s.fail()) cout << s << endl;
40      s = force.add("pthread", &::force);
41      if (s.fail()) cout << s << endl;
42      s = force.set(0, mass);
43      if (s.fail()) cout << "Adding mass failed:" << s << endl;
44      s = force.set(1, "position");
45      if (s.fail()) cout << "Adding position failed:" << s <<
           endl;
46      force.set(2, "velocity");
47      if (s.fail()) cout << "Adding velocity failed:" << s <<
           endl;
48      force.set(3, variant(time_step));
49      if (s.fail()) cout << "Adding timestep failed:" << s <<
           endl;
50
51      while (current_time < end_time)
52      {
53        fos << "Current time: " << current_time << std::endl;
54        s = force.start(distr.units(1));
55        if (s.fail()) break;
56        pos->synchronize();
57
58        current_time += time_step;
59      }
60    } while (false);
61
62    acaf::finalize();
63    MPI_Finalize();
64
```

```
65   if (s.fail())
66     printf("An error %d (%s) occurred. Failed!\n", s.code(), s
         .name());
67   else
68     printf("Success!\n");
69
70   return s.code();
71 }
```

This main function implementation provides the basic necessary code to initialize correctly the environment, to instantiate the entities, to perform the particle system evolving loop and to clean up the objects.

- The initialization step includes 2 function calls: *MPI_Init* and *acaf::initialize* (lines 3 and 4). According to the MPI user manual, the *MPI_Init* should always be the first function call of the application. Therefore, it is impossible to integrate it as a part of the framework initialization.
- The instantiation of the entities includes: the distribution creation using the configuration file (line 11); the contents creation (lines 16, 20 and 27) and initialization - the masses are set to 1; the positions are randomized in the range between $(-1,-1,-1)$ and $(1,1,1)$; the velocities are set to $(0,0,0)$ (lines 19, 24 and 30); the kernel creation and adding the available implementations and arguments (lines 37-49).
- The particle system evolving loop (lines 51-59) consists of synchronous execution of the kernel (line 54), synchronizing the positions (line 56) and proceeding to the next time frame (line 58).
- Finally, the clean up of the environment also includes 2 function calls symmetric to the initialization: *MPI_Finalize* and *acaf::finalize* (lines 62 and 63).

5.1.5 Analysis

The usage example demonstrates the following properties of the framework:

1. the framework provides an ability to separate the data mechanisms from the computational code and the environmental code. The computational code operates with the data using the pointers to some region of memory. Conversely, the environmental code operates with the instances of *Content* subclasses using the convenience functions: *random*, *fill*, *synchronize* and other;
2. the framework encapsulates the device-specific operations: specifying the utilization schema of the device in the configuration file and providing the computational code for the device is enough for including the device in the computation;
3. also, the framework encapsulates the network-specific operations: it is only necessary to mention the network communication library in the configuration file and place the initialization and cleanup function calls (if they are required by the library);

4. the framework targets the data-parallel problems, the task-parallel problem implementation cannot be facilitated by the framework;

5. the framework requires the user to implement the computational code separately for each *Technology* instance used in the configuration. Moreover, it is necessary that the code is reentrant, since it is executed simultaneously on different data.

5.1.6 Test Setup

For testing the framework, the following heterogeneous cluster was used: the 7-nodes cluster with 4 processing nodes, each of them has the NVIDIA GeForce GTX 285 GPU with 2GB of RAM, the Intel Xeon E5504 CPU and 6GB of RAM. The nodes run Debian OS. The nodes are connected using Infiniband interface.

5.2 Benchmarking

To evaluate the framework using some measurable metrics, benchmarking with different parameters was performed. Benchmarking includes executing the usage example with different number of particles, different configurations (with or without some devices, with or without network utilization). The execution times for all different runs are combined in Figures 5.1, 5.2 and 5.3.

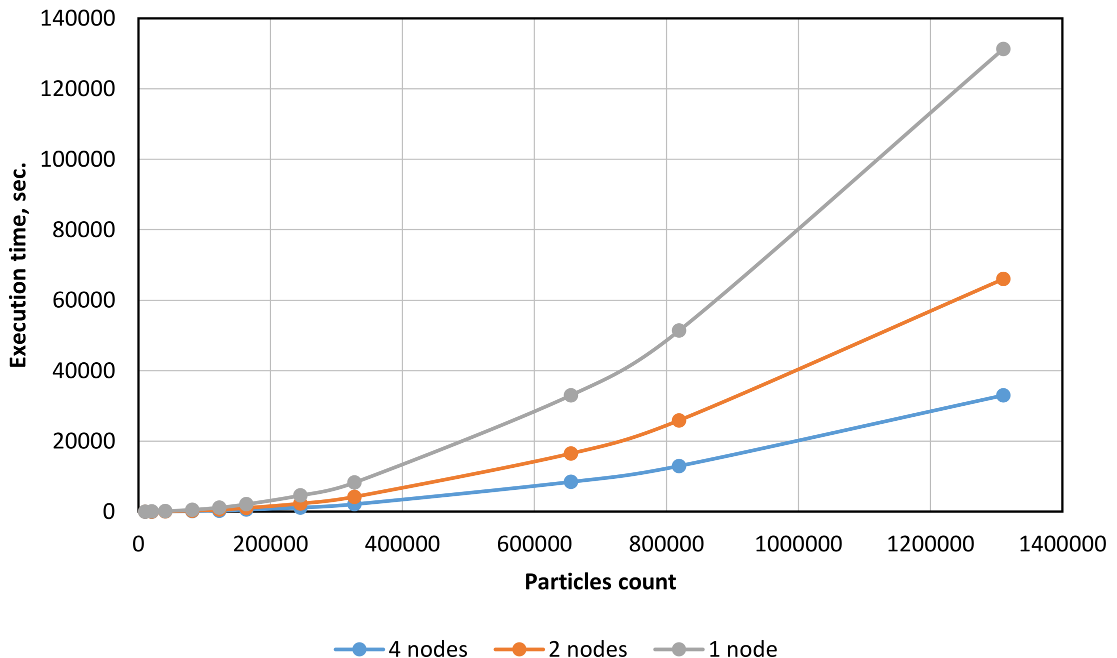

Figure 5.1: The full comparison chart of running the code on a different number of nodes.

The comparison charts show that the most efficient way to run the computation on the heterogeneous cluster using the ACAF is a distributed computation performed on

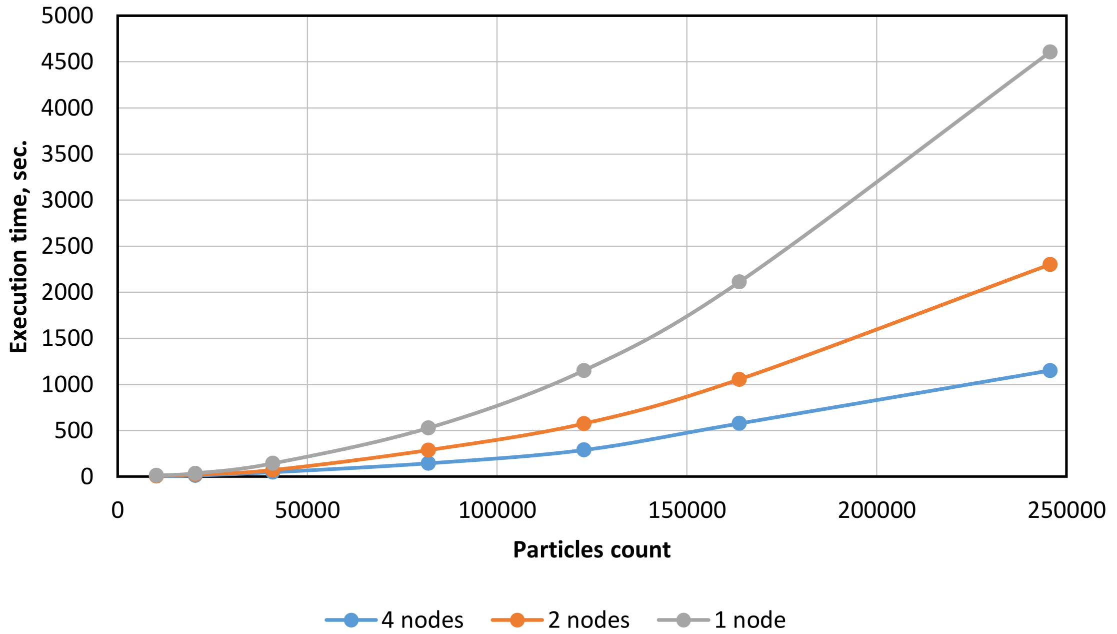

Figure 5.2: The lower range comparison chart of running the code on a different number of nodes.

GPUs only. The deeper analysis of the execution times of the simulation within the different numbers of nodes shows:

- the average ratio of the execution times between 2 nodes configuration and 1 node configuration is 1.971x;
- the average ratio of the execution times between 4 nodes configuration and 2 nodes configuration is 1.97x;
- the average ratio of the execution times between 4 nodes configuration and 1 node configuration is 3.882x.

These ratios are quite near the ideal ratios 2, 2 and 4. This proofs the efficiency of the distribution mechanisms based on the design and implemented in the framework. The average ratios mentioned above consider only the distribution-effective execution times, in particular, the cases with at least 81920 particles. For the lower amount of particles, the network transferring overhead drops the whole performance of the computation.

The comparison of different device configurations (GPU only, CPU and GPU, CPU only) shows that the GPU computation is 40x times faster, than CPU computation. This speed-up factor also explains that combining CPU and GPU computations makes no sense for the lower number of particles being 10x times slower as the GPU computation and 4x faster as the CPU computation.

Additionally, the comparison of the framework performance against the bare code performance was done (the source code of the bare implementation can be found in Appendix A). This comparison shows what is the overhead of using the framework.

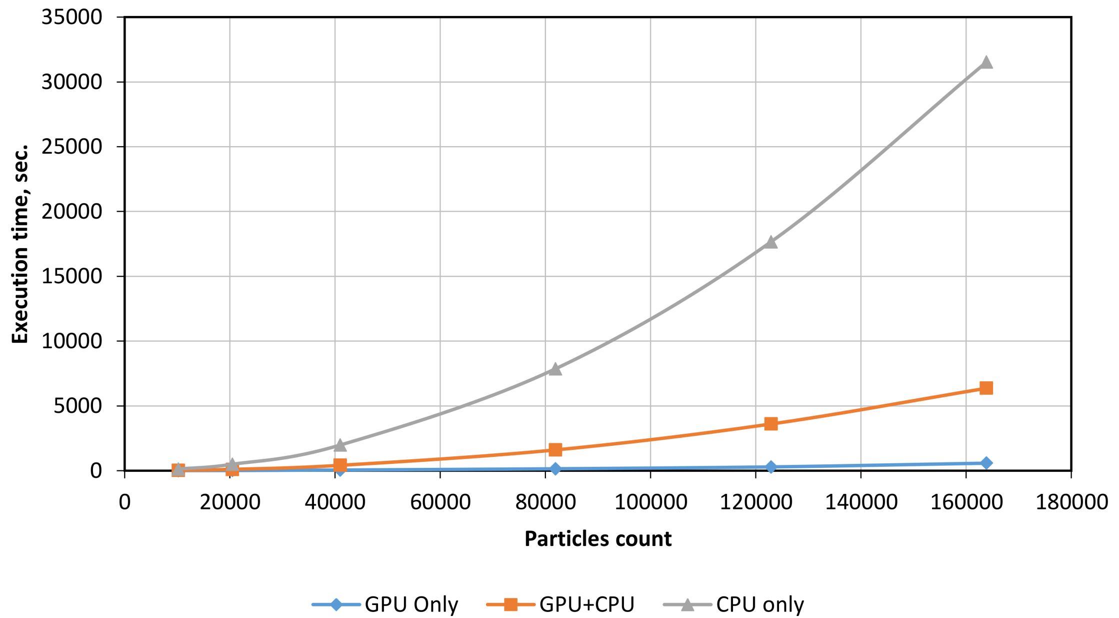

Figure 5.3: The comparison chart of running the code with different hardware configurations.

The bare simulation code consists of the network-distributed computations performed on GPU using the same OpenCL kernel. The full bare simulation code can be found in the Appendix 1. Figure 5.4 represents the percent overhead of the execution time of the usage example to the execution time of the bare implementation scaled over the particles number in the example system.

According to this chart, we can state that the time overhead of using ACAF approximates 0 for the larger particle systems and is equal to 4 seconds for the case of 1310720 particles.

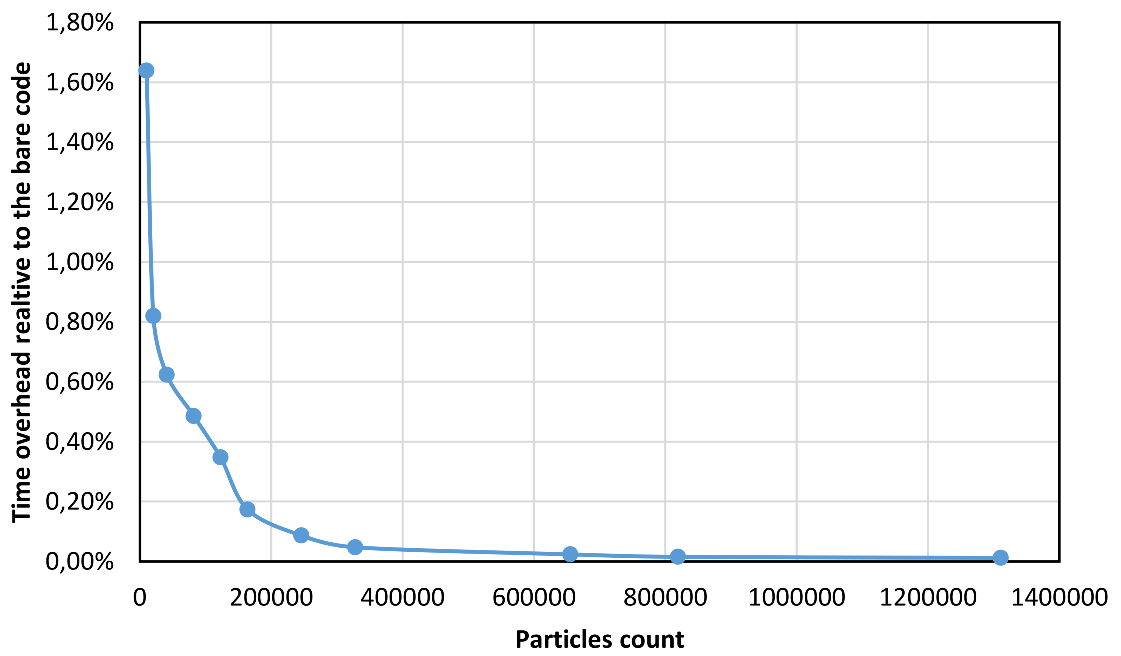

Figure 5.4: The comparison chart of the ACAF-based implementation to the bare implementation.

Chapter 6

Discussion and Conclusion

6.1 Pros and Cons

The following advantages can be mentioned as a result of comparing the final framework design and its implementation with the other approaches mentioned in Section 2.2 and the bare simulation code implementation:

1. The design of the framework prescribes the clear separation of the data mechanisms from the computational code and the environmental code. The data operations are managed by the data-relevant entities of the framework: *Database*, *Storage*, and *Content*. This enables one to encapsulate the necessary complex data operations: distribution of the data, its transferring and its synchronizing (see 1 in Section 5.1.5).

2. The data operations are also separated according to the type of the operation: data allocating and transferring is managed by *Storage* classes (see Subsection 4.2.4.4), data interpretation and logic operations on the data are managed by *Content* classes (see Subsection 4.2.4.6), while the *Database* class guarantees the correct functionality and provides some miscellaneous functions. Such splitting helps to extend only the necessary framework parts.

3. The framework also splits the simulation implementation into several distinct parts, which makes the coding task transparent:

    - the configuration file specifies the devices and nodes to be used and defines the distribution of the data (see Listing 5.1);
    - the kernel implementations represent mathematical and physical parts of the code (see Listings 5.2 and 5.3);
    - the environmental code does the initialization, data definition, data initialization, kernel instantiation and defines the main particle system evaluation loop (see Listing 5.4).

4. The framework is designed as a C++ framework. This means that the user and the framework developer have access to a large range of different powerful system

calls and a variety of computational libraries and tools. So, the user has a choice either to re-implement the algorithm using the framework tools or to reuse the existing solution. Moreover, the availability of the system calls provides an option of performance-targeted tuning of the final application.

5. The framework encapsulates the device-specific operations using the *Architecture* and *Technology* classes (see 2 in Section 5.1.5). The encapsulation of the operations implies that the final user should not know and should not use some device-specific functions, libraries and tools. The framework classes also separate the functional aspects of the work with some device present in the system: *Architecture* class enables the devices of some specific type to be recognized and used by the framework (see Section 4.4.4); *Technology* class focuses on the device utilization schema (see Section 4.4.5). This separation facilitates the possibilities of extending the framework: the developer is able to target one of the aspects (see Section 4.3.1).

6. The framework also encapsulates the network-distributed communications and computations. This encapsulation provides users with an ability to avoid the network-related operations and offers rather the possibility to switch easily between the single-node and multi-nodes configurations (see 3 in Section 5.1.5).

7. The time overhead of using the framework in comparison to the bare code is very low and approximates 0% with the growth of the particles count (see Figure 5.4). The network distribution mechanisms of the framework are efficient and do not introduce an additional overhead as it was shown in Section 5.2.

Meanwhile, the current implementation of the framework has the following limitations and disadvantages:

1. The framework requires the knowledge and usage of the technology-targeted computational languages to utilize the computational devices, such as OpenCL C and CUDA C for utilizing GPUs (as it is shown in the usage example in Section 5.1). The computational kernel used by the framework to execute the actual calculations should be implemented by the user for each type of technology combined in the current computational context. To avoid the necessity of having the individual kernel implementations for each technology, it is vital to design and implement some common parallel programming language, which can be further translated into the technology-specific languages. Still, it is crucial to maintain an ability to use the native technology-specific programming languages in order to be able to finely-tune a particular computational code.

2. In addition to the technology-targeted programming languages requirement, the reentrance of the user-defined computational code is necessary, even though it complicates the implementation of the kernel for different devices. At the same time, the wrongly implemented kernel executed simultaneously on different data can lead to hard-recognizable incorrect results (see 5 in Section 5.1.5).

3. In comparison to the other approaches described in Section 2.2, the proposed framework design still requires some coding work to be done. The amount of coding can be reduced by implementing the DSL as a layer over the framework functionality.

4. The framework exclusively targets the data-parallel problems, particularly the particle problems. The framework does not fit for the task-parallel problems. Utilizing the framework for some other data-parallel problems rather than the particle problems may require an implementation of some other *Content* subclasses (see 4 in Section 5.1.5).

5. There is no possibility to provide immediately some user-driven testing of the framework, since the main advantage of many other approaches (as in e.g. 2.2.4) is the availability of many different ready-to-use modules, the combination of which leads to the necessary solution. This means that the implemented framework misses the set of built-in modules/functions/classes, which will serve the same purpose.

6. Also the chosen programming language C++ is not an optimal one, because most astrophysicists work at the current moment with Fortran90. This means that the actual using of the framework will imply the change of the working programming language.

6.2 Criteria Evaluation

When discussing pros and cons of the framework, it is necessary to evaluate the implemented approach against the criteria described in Chapter 3:

1. As it was shown in Figure 5.4, the time overhead of using the ACAF in comparison to the bare simulation code reaches 1.6% for small particle systems and approaches 0% for larger particle systems. Still, the absolute time overhead for 1310720 particles is 4 seconds.

2. The usage example (see Section 5.1) demonstrated that it is unnecessary to write any device-specific code (device queries, device memory allocations, device data transferring, device communication) in order to execute the simulation on different computational units.

3. Also, the same usage example demonstrated that it is possible to execute the simulation on the heterogeneous cluster without explicitly writing any network-specific code (network queries, network data transferring), with an exception of initialization and cleanup function calls.

4. The concept of the configuration file (see Section 4.3.2 and Listing 5.1) allows one to use the different utilization schemes within the same executable program. Still, adding the new technologies requires the user explicitly to implement the kernels for these technologies.

5. The current implementation of the framework does not cover the uniformity of the computational code. The kernel should be implemented separately for each technology in use (see Listings 5.2 and 5.3). This issue is addressed in future work (see Section 7).

6. The framework is based on the hierarchy of C++ classes, which are responsible for the different aspects of the heterogeneous cluster programming (see Section 4.4). Growing the hierarchy by implementing the new C++ classes allows for the possibility to extend the framework with new functionality.

6.3 Retrospective

The most problematic part of the developed framework is the detached design from the prospective users. The framework is designed without any feedback from astrophysicists. This disadvantage unfortunately results in the absence of user testing at the final stage of the framework implementation. The main reasons of the detached development were the following:

1. The astrophysics are used to work with Fortran90 programming language. But C++ is chosen due to the following facts:
    - C++ programming language as an operating system language for the target platform provides different performance tuning possibilities (using assembler insertions, SSE command, direct operating system calls and hardware calls);
    - C++ has an ability to call the functions from the binary libraries originally written in other programming languages (including Fortran90);
    - C++ functions exported from the binary library as C-style functions can be called directly from Fortran90.

2. The proposed design of the framework implies the need to re-implement the various helper computational functions in order to be able to perform the same simulation tasks as scientists currently target. This re-implementing lies beyond the scientists' interests and the inability to continue the active research with the new framework reduces their willingness to proceed with the design.

Based on the implemented framework, it might be possible to resolve the identified problems which enable the user-driven development of the framework. In particular, the following features could have been integrated:

1. The functions used in the environmental code of the framework could have been implemented together with the C-style interface, which would enable scientists to use Fortran90 at least for the environmental code. Still, the kernel implementation should be written in the technology-targeted language.

2. The reusing of the existing libraries should have been integrated more deeply to enable passing the content memory buffers to the foreign functions.

Chapter 7

Future Work

The future work on the framework can be performed by extending it with the following features:

- The tree-structure content classes which can be directly utilized for advanced SPH and N-Body simulations. Such classes will significantly enhance the usability of the framework. The usage of the octree structures in the particle problems is the effective method in case of a large number of particles.
- The current implementation of the pthread technology provides an ability to implement a kernel within a pointer to the function of the particular semantic. Such usage schema is not optimal for the large projects with many different kernels. Therefore, it makes sense to have the dynamic calls to the functions for the pthread technology. The most reasonable way to implement the dynamic calling to the functions consists of using the third-party library "dyncall". The library encapsulates the dynamic function semantic in addition to encapsulating the dynamic calls. This means that the arguments to the function will be passed directly without wrapping them into the *acaf::variant_vector* collection.
- The actual error handling mechanism relies on the initialization order of the error codes. This means that the particular integer error codes are dynamic and can differ within several runs of the same code on different machines. Such inconstant error codes complicate the integration of the framework into the complex applications, since the client application can not process the errors by the integer codes. Therefore, the error handling should be revised to make the integer error codes more persistent.
- The current content classes provide only the full data range synchronization. This limitation prevents the framework usage for the very large data on the cluster with poor local storage capacities, when the full range of all the necessary data can not be stored at once in the local memory. In this case, the computation of a single iteration is usually split into several steps. To support such processing schema, the framework needs the content classes for the partially synchronized arrays.
- Another useful feature for the framework is the support of astrophysical-native file formats: Hierarchical Data Format version 5 (HDF5), Flexible Image Transport

System (FITS). Such support will make it possible to initialize the data and report the results in the necessary formats without an additional effort from the user.

- Finally, the most valuable modification of the framework lies in designing and implementing of the Domain Specific Language, which is to encapsulate the current numerous framework function calls into the language commands. This modification will make the usage of the framework even more transparent and will significantly decrease the amount of the necessary coding work. Still, the language should keep open an ability to switch to the direct framework calls and to provide the kernel implementation in the technology-native programming languages.

Appendices

Appendix A

N-Body Simulation Code

Listing A.1: "The Main Function"

```
 1 #include <time.h>
 2 #include <stdlib.h>
 3 #include <dirent.h>
 4 #include <sys/stat.h>
 5 #include <errno.h>
 6 #include <stdio.h>
 7 #include <string.h>
 8 #include <ctime>
 9
10 #include <unistd.h>
11 #include <limits.h>
12
13 #include <mpi.h>
14
15 #define RANDOM_PARTICLES 2560
16
17 unsigned int  partNumber    = 0;
18 cl_double    * mass          = NULL;
19 cl_uint      * ids           = NULL;
20 cl_double4   * position      = NULL;
21 cl_double4   * velocity      = NULL;
22
23 void load_data(int mpi_size, int mpi_id)
24 {
25     srand(time(NULL));
26
27     partNumber = mpi_size * RANDOM_PARTICLES;
28
29     mass            = (cl_double *) malloc(partNumber * sizeof(cl_double));
30     ids             = (cl_uint *)   malloc(partNumber * sizeof(cl_uint));
31     position        = (cl_double4 *)malloc(partNumber * sizeof(cl_double4));
32     velocity        = (cl_double4 *)malloc(partNumber * sizeof(cl_double4));
33
34     for (unsigned int i = 0; i < partNumber; ++i)
35     {
36         ids[i] = i + 1;
37         mass[i] = 1.;
38         position[i].s[0] = ((cl_double)rand()) / RAND_MAX;
39         position[i].s[1] = ((cl_double)rand()) / RAND_MAX;
40         position[i].s[2] = ((cl_double)rand()) / RAND_MAX;
41         position[i].s[3] = .0;
42         velocity[i].s[0] = .0;
43         velocity[i].s[1] = .0;
44         velocity[i].s[2] = .0;
45         velocity[i].s[3] = .0;
46     }
47
48     MPI_Allgather(&(position[mpi_id * RANDOM_PARTICLES]), 4 * RANDOM_PARTICLES, MPI_DOUBLE,
          position, 4 * RANDOM_PARTICLES, MPI_DOUBLE, MPI_COMM_WORLD);
49 }
50
51 int main(int argc, char ** argv)
52 {
53     clock_t start = clock();
54
55     MPI_Init(&argc, &argv);
56
57     int mpi_size, mpi_id;
58     MPI_Comm_size(MPI_COMM_WORLD, &mpi_size);
```

```
59     MPI_Comm_rank(MPI_COMM_WORLD, &mpi_id);
60
61     double time_cur, time_step, time_max;
62     time_cur = 0.;
63     time_step = 0.1;
64     time_max = 1.;
65
66     load_data(mpi_size, mpi_id);
67
68     prepareEnvironment();
69
70     transferDataToDevice();
71
72     while (time_cur < time_max)
73     {
74         refine_positions(time_step, mpi_id * RANDOM_PARTICLES, RANDOM_PARTICLES);
75         transferDataToHost(true, mpi_id * RANDOM_PARTICLES, RANDOM_PARTICLES);
76
77         MPI_Allgather(&(position[mpi_id * RANDOM_PARTICLES]), 4 * RANDOM_PARTICLES,
               MPI_DOUBLE, position, 4 * RANDOM_PARTICLES, MPI_DOUBLE, MPI_COMM_WORLD);
78         transferDataToDevice(true);
79
80         time_cur += time_step;
81     }
82
83     freeEnvironment();
84
85     MPI_Finalize();
86
87     clock_t finish = clock();
88
89     printf("Done in %g seconds\n", ((double)(finish-start))/CLOCKS_PER_SEC);
90     return 0;
91 }
```

Listing A.2: "The GPU Management Code"

```
 1 #define FAILED(errcode) (CL_SUCCESS != errcode)
 2 #define ITEMS_PER_GROUP 256
 3
 4 size_t              wgrNum            = 0;
 5 size_t              witNum            = 0;
 6 size_t              wgrSize           = 0;
 7
 8 const char *        g_stProgramFile   = "gravity.cl";
 9 const char *        g_stAccCalcKern   = "force";
10 const char *        g_stPosRefineKern = "refine_positions";
11 const char *        g_stBuildOptions  = "-cl-mad-enable";// -cl-nv-maxrregcount=20";
12 const char *        g_stBuildKernelLogName = "BuildKernelLog.txt";
13
14 cl_platform_id *    cpPlatformsList = NULL;
15 cl_device_id   *    cdDevicesList   = NULL;
16 cl_context          ccContext       = NULL;
17 cl_command_queue    cqKernelQueue   = NULL;
18 cl_command_queue    cqTransferQueue = NULL;
19
20 cl_program          cpProgram       = NULL;
21 cl_kernel           ckAccCalc       = NULL;
22 cl_kernel           ckPosRefine     = NULL;
23
24 cl_mem              cmMassBuf       = NULL;
25 cl_mem              cmPositionBuf   = NULL;
26 cl_mem              cmVelocityBuf   = NULL;
27
28 void refine_positions(double time_step, int mpi_offset, size_t size_per_process)
29 {
30     clSetKernelArg(ckAccCalc, 3, sizeof(double), &time_step);
31     cl_uint4 actual;
32     actual.s[0] = witNum;
33     actual.s[1] = actual.s[2] = actual.s[3] = 0;
34     clSetKernelArg(ckAccCalc, 4, sizeof(cl_uint4), &actual);
35     size_t off = mpi_offset;
36     clEnqueueNDRangeKernel(cqKernelQueue, ckAccCalc, 1, &off, &size_per_process, &wgrSize, 0,
           NULL, NULL);
37     clFinish(cqKernelQueue);
38 }
39
40 void transferDataToHost(bool minimal, int offset, int size)
41 {
42     clFinish(cqKernelQueue);
43     if (size == -1 || !minimal)
44     {
45       offset = 0;
46       size = partNumber;
47     }
48
```

```
49      clEnqueueReadBuffer(cqTransferQueue, cmPositionBuf, CL_TRUE, sizeof(cl_double4) * offset,
             sizeof(cl_double4) * size, &(position[offset]), 0, NULL, NULL);
50      if (!minimal)
51      {
52        clEnqueueReadBuffer(cqTransferQueue, cmVelocityBuf, CL_TRUE, 0, sizeof(cl_double4) *
              partNumber, velocity, 0, NULL, NULL);
53      }
54  }
55
56  void transferDataToDevice(bool minimal)
57  {
58    if (minimal)
59    {
60      clEnqueueWriteBuffer(cqTransferQueue, cmPositionBuf, CL_TRUE, 0, sizeof(cl_double4) *
             partNumber, position, 0, NULL, NULL);
61    }
62    else
63    {
64      void * temp = malloc(sizeof(cl_double4) * ITEMS_PER_GROUP);
65      memset(temp, 0, sizeof(cl_double4) * ITEMS_PER_GROUP);
66
67      clEnqueueWriteBuffer(cqTransferQueue, cmMassBuf,     CL_TRUE, 0, sizeof(cl_double)  *
             partNumber, mass,     0, NULL, NULL);
68      clEnqueueWriteBuffer(cqTransferQueue, cmPositionBuf, CL_TRUE, 0, sizeof(cl_double4) *
             partNumber, position, 0, NULL, NULL);
69      clEnqueueWriteBuffer(cqTransferQueue, cmVelocityBuf, CL_TRUE, 0, sizeof(cl_double4) *
             partNumber, velocity, 0, NULL, NULL);
70
71      if (witNum > partNumber)
72      {
73          clEnqueueWriteBuffer(cqTransferQueue, cmMassBuf,     CL_TRUE, sizeof(cl_double)  *
                 partNumber, sizeof(cl_double)  * (witNum - partNumber), temp, 0, NULL, NULL);
74          clEnqueueWriteBuffer(cqTransferQueue, cmPositionBuf, CL_TRUE, sizeof(cl_double4) *
                 partNumber, sizeof(cl_double4) * (witNum - partNumber), temp, 0, NULL, NULL);
75          clEnqueueWriteBuffer(cqTransferQueue, cmVelocityBuf, CL_TRUE, sizeof(cl_double4) *
                 partNumber, sizeof(cl_double4) * (witNum - partNumber), temp, 0, NULL, NULL);
76      }
77    }
78  }
79
80  bool prepareEnvironment()
81  {
82      cl_int errorCode = CL_SUCCESS;
83
84      wgrSize = ITEMS_PER_GROUP;
85      witNum = partNumber + ((partNumber % wgrSize == 0)?0:wgrSize) - (partNumber % wgrSize);
86      wgrNum = witNum / wgrSize;
87
88      do {
89          // Queries all possible platforms
90          cl_uint uiNumPlatforms = 0;
91          errorCode = clGetPlatformIDs(0, NULL, &uiNumPlatforms);
92          if (FAILED(errorCode)) break;
93
94          cpPlatformsList = (cl_platform_id *) malloc(sizeof(cl_platform_id) * uiNumPlatforms);
95          errorCode = clGetPlatformIDs(uiNumPlatforms, cpPlatformsList, NULL);
96          if (FAILED(errorCode)) break;
97
98          // Retrieves the information about devices
99          cl_uint uiNumDevices = 0;
100         errorCode = clGetDeviceIDs(cpPlatformsList[0], CL_DEVICE_TYPE_GPU, 0, NULL, &
                 uiNumDevices);
101         if (FAILED(errorCode)) break;
102
103         cdDevicesList = (cl_device_id *)malloc(sizeof(cl_device_id) * uiNumDevices);
104         errorCode = clGetDeviceIDs(cpPlatformsList[0], CL_DEVICE_TYPE_GPU, uiNumDevices,
                 cdDevicesList, NULL);
105         if (FAILED(errorCode)) break;
106
107         char devname[1000] = {0};
108         clGetDeviceInfo(cdDevicesList[0], CL_DEVICE_NAME, 1000, devname, nullptr);
109         printf("Using the device: %s\n", devname);
110
111         ccContext = clCreateContext(NULL, 1, cdDevicesList, NULL, NULL, &errorCode);
112         if (FAILED(errorCode)) break;
113
114         cmMassBuf       = clCreateBuffer(ccContext, CL_MEM_READ_WRITE,  sizeof(cl_double)  *
                 witNum, NULL, &errorCode);
115         if (FAILED(errorCode)) break;
116
117         cmPositionBuf   = clCreateBuffer(ccContext, CL_MEM_READ_WRITE, sizeof(cl_double4) *
                 witNum, NULL, &errorCode);
118         if (FAILED(errorCode)) break;
119
120         cmVelocityBuf   = clCreateBuffer(ccContext, CL_MEM_READ_WRITE, sizeof(cl_double4) *
                 witNum, NULL, &errorCode);
121         if (FAILED(errorCode)) break;
```

```
122
123         cqKernelQueue = clCreateCommandQueue(ccContext, cdDevicesList[0],
                CL_QUEUE_OUT_OF_ORDER_EXEC_MODE_ENABLE, &errorCode);
124         if (FAILED(errorCode)) break;
125
126         cqTransferQueue = clCreateCommandQueue(ccContext, cdDevicesList[0],
                CL_QUEUE_OUT_OF_ORDER_EXEC_MODE_ENABLE, &errorCode);
127         if (FAILED(errorCode)) break;
128
129         FILE* fProgram = NULL;
130
131         fProgram = fopen(g_stProgramFile, "rb");
132         if (NULL == fProgram)
133         {
134             errorCode = CL_INVALID_PROGRAM;
135             break;
136         }
137
138         // get the length of the source code
139         fseek(fProgram, 0, SEEK_END);
140         size_t szSourceLength = ftell(fProgram);
141         fseek(fProgram, 0, SEEK_SET);
142
143         // allocate a buffer for the source code string and read it in
144         char * stSource = (char *)malloc(szSourceLength + 1);
145         if (fread(stSource, szSourceLength, 1, fProgram) != 1)
146         {
147             fclose(fProgram);
148             free(stSource);
149             errorCode = CL_INVALID_PROGRAM;
150             break;
151         }
152
153         fclose(fProgram);
154         stSource[szSourceLength] = '\0';
155
156         cpProgram = clCreateProgramWithSource(ccContext, 1, (const char **)&stSource, &
                szSourceLength, &errorCode);
157         errorCode = clBuildProgram(cpProgram, 1, cdDevicesList, g_stBuildOptions, NULL, NULL)
                ;
158         if (FAILED(errorCode))
159         {
160             // Retrieves the detailed build log in the case of build error
161             FILE *fBuildLog = fopen(g_stBuildKernelLogName, "w");
162
163             if (NULL != fBuildLog)
164             {
165                 size_t szBuildLogSize = 0;
166                 clGetProgramBuildInfo(cpProgram, cdDevicesList[0], CL_PROGRAM_BUILD_LOG, 0,
                        NULL, &szBuildLogSize);
167                 char *buf = (char *)malloc(sizeof(char) * (szBuildLogSize + 1));
168                 memset(buf, 0, sizeof(char) * (szBuildLogSize + 1));
169                 clGetProgramBuildInfo(cpProgram, cdDevicesList[0], CL_PROGRAM_BUILD_LOG,
                        szBuildLogSize, buf, NULL);
170
171                 fprintf(fBuildLog, buf);
172
173                 fclose(fBuildLog);
174             }
175
176             break;
177         }
178
179         ckAccCalc = clCreateKernel(cpProgram, g_stAccCalcKern, &errorCode);
180         if (FAILED(errorCode)) break;
181
182         clSetKernelArg(ckAccCalc, 0, sizeof(cl_mem), &cmMassBuf);
183         clSetKernelArg(ckAccCalc, 1, sizeof(cl_mem), &cmPositionBuf);
184         clSetKernelArg(ckAccCalc, 2, sizeof(cl_mem), &cmVelocityBuf);
185
186     } while (false);
187
188     if (FAILED(errorCode))
189         printf("Some error occurred during initialization %d\n", errorCode);
190
191     return !(FAILED(errorCode));
192 }
193
194 void freeEnvironment()
195 {
196     if (!!ckAccCalc)        clReleaseKernel(ckAccCalc), ckAccCalc = NULL;
197     if (!!cpProgram)        clReleaseProgram(cpProgram), cpProgram = NULL;
198
199     if (!!cqTransferQueue)  clReleaseCommandQueue(cqTransferQueue), cqTransferQueue = NULL;
200     if (!!cqKernelQueue)    clReleaseCommandQueue(cqKernelQueue),   cqKernelQueue = NULL;
201
202     if (!!cmMassBuf)        clReleaseMemObject(cmMassBuf), cmMassBuf = NULL;
```

```
203     if (!!cmPositionBuf)      clReleaseMemObject(cmPositionBuf), cmPositionBuf = NULL;
204     if (!!cmVelocityBuf)      clReleaseMemObject(cmVelocityBuf), cmVelocityBuf = NULL;
205 
206     if (!!ccContext)          clReleaseContext(ccContext), ccContext = NULL;
207 
208     if (!!cpPlatformsList)    free(cpPlatformsList), cpPlatformsList = NULL;
209     if (!!cdDevicesList)      free(cdDevicesList), cdDevicesList = NULL;
210 }
```

Listing A.3: "OpenCL GPU Kernel"

```
1 #pragma OPENCL EXTENSION cl_khr_fp64 : enable
2 
3 #define SOFTENING 0.001
4 
5 #ifndef ITEMS_PER_GROUP
6 #define ITEMS_PER_GROUP 256
7 #endif
8 
9 __kernel void force
10 (
11     __global double  * mass,
12     __global double4 * pos,
13     __global double4 * velo,
14     double time_step,
15     uint4 global_size
16 )
17 {
18     __local double4 shared_this_pos[ITEMS_PER_GROUP];
19 
20     size_t lid = get_local_id(0);
21 
22     barrier(CLK_GLOBAL_MEM_FENCE);
23     shared_this_pos[lid] = pos[get_global_id(0)];
24     double4 this_acc;
25 
26     this_acc.x = this_acc.y = this_acc.z = this_acc.w = .0;
27 
28     for (unsigned int i = 0; i < global_size.x; ++i)
29     {
30         double4 dist_vec = shared_this_pos[lid] - pos[i];
31         this_acc += mass[i] * dist_vec / powr(length(dist_vec) + SOFTENING, 3.);
32     }
33 
34     size_t gid = get_global_id(0);
35     size_t vid = gid - get_global_offset(0);
36 
37     barrier(CLK_GLOBAL_MEM_FENCE);
38     velo[vid] += this_acc * time_step;
39     barrier(CLK_GLOBAL_MEM_FENCE);
40     pos[gid] += velo[vid] * time_step;
41 }
```

List of Figures

1.1 The serial algorithm of the N-Body force simulation. 17
1.2 The parallel algorithm of the N-Body force simulation. 19
1.3 The network-enabled algorithm of the N-Body force simulation. 20
1.4 The heterogeneous algorithm of the N-Body force simulation. 22

4.1 The 3-concepts design. 40
4.2 Diagram of the ACAF database design. 45
4.3 Diagram of the ACAF framework design. 46
4.4 The overview UML diagram of key classes of ACAF. 53
4.5 The UML diagram of Architecture subclasses. 56
4.6 The UML diagram of the Technology subclasses. 59

5.1 The full comparison chart of running the code on a different number of nodes. 75
5.2 The lower range comparison chart of running the code on a different number of nodes. 76
5.3 The comparison chart of running the code with different hardware configurations. 77
5.4 The comparison chart of the ACAF-based implementation to the bare implementation. 78

Bibliography

[1] CLara.
URL `https://www.alpha-tierchen.de/~bkoenig/clara/`

[2] The Enzo Project.
URL `http://enzo-project.org/`

[3] Khronos Group - OpenCL.
URL `http://www.khronos.org/opencl`

[4] Khronos Group - SyCL.
URL `http://www.khronos.org/sycl`

[5] libconfig – C/C++ Configuration File Library.
URL `http://www.hyperrealm.com/libconfig/`

[6] TOP500.
URL `https://en.wikipedia.org/wiki/TOP500`

[7] A. S. A. Barak. The Virtual OpenCL (VCL) Cluster Platform. In *Proc. Intel European Research & Innovation Conference*, page 196, 2011.

[8] A. Ahmad and L. Cohen. A numerical integration scheme for the N-body gravitational problem. volume 12(3):pages 389–402, July 1973. ISSN 0021-9991 (print), 1090-2716 (electronic). doi: http://dx.doi.org/10.1016/0021-9991(73)90160-5.

[9] I. Antcheva, M. Ballintijn, B. Bellenot, M. Biskup, R. Brun, N. Buncic, P. Canal, D. Casadei, O. Couet, V. Fine, L. Franco, G. Ganis, A. Gheata, D. G. Maline, M. Goto, J. Iwaszkiewicz, A. Kreshuk, D. M. Segura, R. Maunder, L. Moneta, A. Naumann, E. Offermann, V. Onuchin, S. Panacek, F. Rademakers, P. Russo, and M. Tadel. {ROOT} - A C++ framework for petabyte data storage, statistical analysis and visualization. *Computer Physics Communications*, volume 180(12):pages 2499 – 2512, 2009. ISSN 0010-4655. doi: http://dx.doi.org/10.1016/j.cpc.2009.08.005. 40 {YEARS} {OF} CPC: A celebratory issue focused on quality software for high performance, grid and novel computing architectures.

[10] C. Arasa, M. C. van Hemert, E. F. van Dishoeck, and G. J. Kroes. Molecular Dynamics Simulations of CO2 Formation in Interstellar Ices. *The Journal of Physical Chemistry A*, volume 117(32):pages 7064–7074, 2013. doi: 10.1021/jp400065v. PMID: 23550656.

[11] J. Bezanson, A. Edelman, S. Karpinski, and V. B. Shah. Julia: A Fresh Approach to Numerical Computing. *CoRR*, volume abs/1411.1607, 2014.

[12] S. Braibant, G. Giacomelli, and M. Spurio. *Particles and fundamental interactions: an introduction to particle physics.* Springer, 2nd edition, 2011. ISBN 9789400724631.

[13] B. L. Chamberlain. Chapel (Cray Inc. HPCS Language). In D. A. Padua, editor, *Encyclopedia of Parallel Computing*, pages 249–256. Springer, 2011. ISBN 978-0-387-09765-7.

[14] P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. von Praun, and V. Sarkar. X10: An Object-oriented Approach to Non-uniform Cluster Computing. *SIGPLAN Not.*, volume 40(10):pages 519–538, October 2005. ISSN 0362-1340. doi: 10.1145/1103845.1094852.

[15] L. Dagum and R. Menon. OpenMP: An Industry-Standard API for Shared-Memory Programming. *IEEE Comput. Sci. Eng.*, volume 5(1):pages 46–55, January 1998. ISSN 1070-9924. doi: 10.1109/99.660313.

[16] S. Dindar, E. B. Ford, M. Juric, Y. I. Yeo, J. Gao, A. C. Boley, B. Nelson, and J. Peters. Swarm-NG: a CUDA Library for Parallel n-body Integrations with focus on Simulations of Planetary Systems. *CoRR*, volume abs/1208.1157, 2012.

[17] A. Dubey, A. Almgren, J. Bell, M. Berzins, S. Brandt, G. Bryan, P. Colella, D. Graves, M. Lijewski, F. Löffler, B. O'Shea, E. Schnetter, B. V. Straalen, and K. Weide. A survey of high level frameworks in block-structured adaptive mesh refinement packages. *Journal of Parallel and Distributed Computing*, 2014.

[18] A. Dubey, K. Antypas, A. C. Calder, B. Fryxell, D. Q. Lamb, P. M. Ricker, L. B. Reid, K. Riley, R. Rosner, A. Siegel, F. X. Timmes, N. Vladimirova, and K. Weide. The software development process of FLASH, a multiphysics simulation code. In J. Carver, editor, *SE-CSE@ICSE*, pages 1–8. IEEE, 2013. ISBN 978-1-4673-6261-0.

[19] W. Dubitzky, K. Kurowski, and B. Schott, editors. *Large-Scale Computing.* John Wiley & Sons, Inc., Hoboken, NJ, USA, November 2011. ISBN 9781118130506. doi: 10.1002/9781118130506.

[20] C. Feichtinger, S. Donath, H. Köstler, J. Götz, and U. Rüde. WaLBerla: HPC software design for computational engineering simulations. *Journal of Computational Science*, volume 2(2):pages 105–112, May 2011. ISSN 18777503. doi: 10.1016/j.jocs.2011.01.004.

[21] D. Frenkel and B. Smit. *Understanding Molecular Simulation (Second Edition).* Academic Press, second edition edition, 2002. ISBN 978-0-12-267351-1. doi: http://dx.doi.org/10.1016/B978-012267351-1/50003-1.

[22] B. Fryxell, K. Olson, P. Ricker, F. X. Timmes, M. Zingale, D. Q. Lamb, P. MacNeice, R. Rosner, J. W. Truran, and H. Tufo. FLASH: An Adaptive Mesh Hydrodynamics Code for Modeling Astrophysical Thermonuclear Flashes. *Astrophys. J. Supp.*, volume 131:pages 273–334, November 2000. doi: 10.1086/317361.

[23] R. A. Gingold and J. J. Monaghan. Smoothed particle hydrodynamics - Theory and application to non-spherical stars. *Monthly Notices of the Royal Astronomical Society*, volume 181:pages 375–389, November 1977. doi: 10.1093/mnras/181.3.375.

[24] T. Goodale, G. Allen, G. Lanfermann, J. Massó, T. Radke, E. Seidel, and J. Shalf. The Cactus Framework and Toolkit: Design and Applications. In *Vector and Parallel Processing – VECPAR'2002, 5th International Conference, Lecture Notes in Computer Science*. Springer, Berlin, 2003.

[25] A. Graps. N-Body / Particle Simulation Methods, 1996.
URL `http://www.cs.cmu.edu/afs/cs/academic/class/15850c-s96/www/nbody.html`

[26] W. Gropp, E. Lusk, and A. Skjellum. *Using MPI (2Nd Ed.): Portable Parallel Programming with the Message-passing Interface.* MIT Press, Cambridge, MA, USA, 1999. ISBN 0-262-57132-3.

[27] T. Hamada and K. Nitadori. 190 TFlops Astrophysical N-body Simulation on a Cluster of GPUs. In *2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, November, pages 1–9. IEEE, November 2010. ISBN 978-1-4244-7557-5. doi: 10.1109/SC.2010.1.

[28] S. Harfst, A. Gualandris, D. Merritt, R. Spurzem, S. P. Zwart, and P. Berczik. Performance analysis of direct N-body algorithms on special-purpose supercomputers. *New Astronomy*, volume 12(5):pages 357–377, July 2007. ISSN 13841076. doi: 10.1016/j.newast.2006.11.003.

[29] P. Jetley, L. Wesolowski, F. Gioachin, L. V. KalÃ©, and T. R. Quinn. Scaling Hierarchical N-body Simulations on GPU Clusters. In *SC*, pages 1–11. IEEE, 2010. ISBN 978-1-4244-7559-9.

[30] A. Lazzaro, S. Jarp, J. Leduc, A. Nowak, and L. Valsan. Report on the parallelization of the MLfit benchmark using OpenMP and MPI. Technical Report CERN-OPEN-2014-030, CERN, Geneva, Jul 2012.

[31] L. B. Lucy. A numerical approach to the testing of the fission hypothesis. *Astronomical Journal*, volume 82:pages 1013–1024, December 1977. doi: 10.1086/112164.

[32] J. Makino, T. Fukushige, M. Koga, and K. Namura. GRAPE-6: Massively-Parallel Special-Purpose Computer for Astrophysical Particle Simulations. *Publications of the Astronomical Society of Japan*, volume 55(6):pages 1163–1187, 2003. doi: 10.1093/pasj/55.6.1163.

[33] G. Marcus. *Acceleration of Astrophysical Simulations with Special Hardware*. Ph.D. thesis, University of Heidelberg, Germany, 2011.

[34] S. Marri and S. D. M. White. Smoothed particle hydrodynamics for galaxy-formation simulations: improved treatments of multiphase gas, of star formation and of supernovae feedback. *Monthly Notices of the Royal Astronomical Society*, volume 345(2):pages 561–574, 2003. doi: 10.1046/j.1365-8711.2003.06984.x.

[35] N. Nakasato, G. Ogiya, Y. Miki, M. Mori, and K. Nomoto. Astrophysical Particle Simulations on Heterogeneous CPU-GPU Systems, June 2012.

[36] K. M. Pontoppidan, C. P. Dullemond, E. F. van Dishoeck, G. A. Blake, A. C. A. Boogert, N. J. Evans, II, J. E. Kessler-Silacci, and F. Lahuis. Ices in the Edge-on Disk CRBR 2422.8-3423: Spitzer Spectroscopy and Monte Carlo Radiative Transfer Modeling. *The Astrophysical Journal*, volume 622:pages 463–481, March 2005. doi: 10.1086/427688.

[37] D. Razmyslovich and G. Marcus. Astrophysical-oriented Computational multi-Architectural Framework: Design and Implementation. *International Journal On Advances in Intelligent Systems*, volume 9(3&4), forthcoming.

[38] D. Razmyslovich, G. Marcus, and R. Männer. Towards an Astrophysical-oriented Computational multi-Architectural Framework. In *Computational World 2016*, pages 16 – 26. IARIA, 2016. ISSN 2308-4170.

[39] D. Sahu, A. Das, L. Majumdar, and S. K. Chakrabarti. Monte Carlo simulation to investigate the formation of molecular hydrogen and its deuterated forms. *New Astronomy*, volume 38:pages 23 – 30, 2015. ISSN 1384-1076. doi: http://dx.doi.org/10.1016/j.newast.2014.12.011.

[40] V. Springel. The cosmological simulation code GADGET-2. *Monthly Notices of the Royal Astronomical Society*, volume 364:pages 1105–1134, December 2005. doi: 10.1111/j.1365-2966.2005.09655.x.

[41] R. Spurzem, P. Berczik, I. Berentzen, K. Nitadori, T. Hamada, G. Marcus, A. Kugel, R. Männer, J. Fiestas, R. Banerjee, and R. Klessen. Astrophysical particle simulations with large custom GPU clusters on three continents. *Computer Science - Research and Development*, volume 26(3-4):pages 145–151, April 2011. ISSN 1865-2034. doi: 10.1007/s00450-011-0173-1.

[42] T. Stroman. *Particle-in-cell simulation of astrophysical plasmas: probing the origin of cosmic rays*. Ph.D. thesis, Iowa State University, United States of America, 2010.

[43] H. Wang, S. Potluri, M. Luo, A. Singh, S. Sur, and D. Panda. MVAPICH2-GPU: optimized GPU to GPU communication for InfiniBand clusters. *Computer Science - Research and Development*, volume 26(3-4):pages 257–266, 2011. ISSN 1865-2034. doi: 10.1007/s00450-011-0171-3.

[44] L. Wang, R. Spurzem, S. Aarseth, M. Giersz, A. Askar, P. Berczik, T. Naab, R. Schadow, and M. Kouwenhoven. The DRAGON simulations: globular cluster evolution with a million stars. *Monthly Notices of the Royal Astronomical Society*, volume 458(2):pages 1450–1465, 2016.

[45] L. Wang, R. Spurzem, S. Aarseth, K. Nitadori, P. Berczik, M. B. N. Kouwenhoven, and T. Naab. NBODY6++GPU: ready for the gravitational million-body problem. *Monthly Notices of the Royal Astronomical Society*, volume 450(4):pages 4070–4080, 2015. doi: 10.1093/mnras/stv817.

[46] M. Wetzstein, A. F. Nelson, T. Naab, and A. Burkert. Vine - A Numerical Code for Simulating Astrophysical Systems Using Particles. I. Description of the Physics and the Numerical Methods. *The Astrophysical Journal Supplement*, volume 184:pages 298–325, October 2009. doi: 10.1088/0067-0049/184/2/298.

[47] S. P. Zwart. The Astronomical Multipurpose Software Environment and the Ecology of Star Clusters. In *CCGRID*, page 202. IEEE Computer Society, 2013. ISBN 978-1-4673-6465-2.

Acknowledgements

.

To my supervisor Prof. Dr. Reinhard Männer, for giving me a chance to work on this thesis, for his support and guidance during the last seven years.

To Dr. Guillermo Aníbal Marcus Martínez, for his guidance in the beginning of my work on the thesis and for his very valuable comments and advises on the final stages.

To German Academic Exchange Service (DAAD), for providing the financial support during my first year of study.

To the company Volume Graphics GmbH, specially to the directors Christoph Poliwoda, Thomas Günther and Christof Reinhart, for their understanding during my part-time work in the company last three years.

To my parents, for their love and support throughout my life.

To my wife, for everyday inspiration, for her support from the very beginning and for all the good times that keep me going.

To all of them, I thank you sincerely for your support during the time that has taken me to write this thesis.