Astrophysical-oriented Computational multi-Architectural Framework: Design and Implementation

Dzmitry Razmyslovich
Institute for Computer Engineering (ZITI),
University of Heidelberg,
Mannheim, Germany
Email: dzmitry.razmyslovich@ziti.uni-heidelberg.de

Guillermo Marcus
NVIDIA Corporation,
Berlin, Germany
Email: gmarcus@nvidia.com

***Abstract*—In this paper, we present the design details and the implementation aspects of the framework for simplifying software development in the astrophysical simulations branch - Astrophysical-oriented Computational multi-Architectural Framework (ACAF). This paper covers the design decisions involved in reaching the necessary level of abstraction as well as establishing the essential set of objects and functions covering some aspects of application development for astrophysical problems. The implementation details explain the programming mechanisms used and the key objects and interfaces of the framework. 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 approximating to 0% for bigger particle systems. At the same time, the execution with different cluster configurations displays that the program performance scales almost according to the number of cluster nodes in use. These prove the efficiency and usability of the framework implementation.**

***Keywords*–Astrophysics; Heterogeneous; Framework; Cluster; GPGPU.**

I. Introduction

Astrophysical simulation tasks have usually high computational density, therefore it is common to use hardware accelerators for solving them. Also, the astrophysical simulations have a huge amount of data to calculate, which makes it reasonable to use computer clusters. But the data dependencies of the simulation algorithms limit the usage of big clusters because of high data communication rate. Therefore, the astrophysical simulations tasks are normally solved using heterogeneous clusters [1][2][3][4]. According to TOP500, the top-rated heterogeneous clusters use Graphics Processing Units (GPU) or Field Programmable Gate Arrays (FPGA) as computational accelerators.

The most important computational astrophysical problems include N-Body simulations, Smoothed Particle Hydrodynamics (SPH), Particle-Mesh and Radiative Transfer. All of them are usually approximated for the calculation purposes with respective particle physics problems. Where particle physics is a branch of physics which deals with existence and interactions of particles, that refer to some matter or radiation. Therefore, computational astrophysics data represents a collection of particles - a particle system. Each particle contains a number of parameters like position in 3D space, speed, direction, mass, etc. A collection of certain values for all parameters of all particles is named a state of a particle system. At the same time, the computational tasks embrace numerical solving of a number of equations, which evaluate the state of a particle system [5].

This means, astrophysicists should deal a lot with developing simulation programs capable to run on heterogeneous clusters. This requires certain expertise in the following subjects:

- astrophysics, since the problem consists of simulating the astrophysical objects;
- network programming for cluster utilizing;
- parallel programming and hardware accelerators programming including the usage of specific interfaces and languages;
- micro-electronics for designing FPGA boards.

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

Technically, developing of a distributed multi-architectural application could be divided into a set of the following aspects:

- balance loading;
- data communication between nodes;
- data communication between the devices inside of each node;
- computational interfaces for different architectures;
- programming languages for different interfaces (like Open Multi-Processing (OpenMP) for Central Processing Unit (CPU); Open Computing Language (OpenCL), Compute Unified Device Architecture (CUDA), Open Accelerators (OpenACC) for GPU and Very High Speed Integrated Circuit (VHSIC) Hardware Description Language (VHDL) for FPGAs).

All these aspects should be taken into account in order to develop an application and all of them should be examined for the current system in order to reach high computational performance. Hence, it makes sense to have the ACAF, which facilitates astrophysical research by providing the user with a set of objects and functions fulfilling 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 fulfill the previous requirements, the language should have as less run-time expenses (such as using Virtual Machines and etc) as possible, ideally the language should be similar to the one used by scientists at the current moment;
- finally, it would be an additional advantage to preserve a possibility to reuse the existing computational libraries.

The rest of the paper is divided into 7 sections. Section II highlights the currently existing standards, frameworks and languages for the software development targeted heterogeneous systems. Section III consists of 3 subsections, each of them presenting some design motivations and solutions we have used to reach the goal. Sections IV and V uncover some implementation details of the framework. In Section VI, the usage example of the current framework implementation is given. Section VII describes the benchmarking performed to evaluate the result framework. Finally, Section VIII concludes the paper with an outlook to the most important advantages of the ACAF.

II. Current State of Art

This section covers mostly used and important frameworks, libraries, languages and standards, which can optimize or simplify development of the specific astrophysical cluster applications.

A. Standards

This section gives an overview of the currently used programming standards and standard 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.

- MPI [6] 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 a lot of collective data functions, like 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, parallel files handling (including HDF5 files) etc. The most interesting extension in terms of heterogeneous clusters programming is MVAPICH2.
- MVAPICH2 [7] 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. And if the calculations should be done simultaneously on GPU and CPU or several GPU, MVAPICH2 can only be utilized by dividing the calculation on different devices of the same host into separate MPI processes. Another limitation of MVAPICH2 lies in supporting only 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. At the same time, the user should take care of allocation, distribution of the data, code execution, synchronization and so on. Therefore, these libraries cannot 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.
- CUDA stands for Compute Unified Device Architecture 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 the user to utilize GPU 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 GPU 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.

- OpenMP [8] 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. Conversely, the 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.
- OpenACC is a standard for the programming of computational accelerators originally proposed by Nvidia (currently only CUDA-enabled GPU are supported). OpenACC uses the similar API as OpenMP and is also based on the preprocessor pragmas. Additionally 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.
- 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. OpenMP API model definitely reduces the requirements in parallel programming skills abstracting the numerous function calls in easy-readable pragmas. Still, this model does not hide a lot of implementation details, which are out-of-interests for scientific programmers: device data allocation, data transferring, runtime synchronization, etc. On the other side, 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 the 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.
- OpenCL [9] is an open standard for general purpose parallel programming across different heterogeneous processing platforms: CPU, GPU 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 compila-

tion, uploading and execution.

And 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 GPU and other different accelerators such as DSPs (Digital Signal Processors), FPGAs (Field-Programmable Gate Arrays) etc. 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 [10], the project is stalled at OpenCL 1.0), 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 in 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 fulfills 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.

- SyCL [11] 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.

B. *Libraries, Frameworks and Languages*

This section covers numerous libraries and frameworks used or possible to be used for solving the code complexity problem of heterogeneous applications. [12]

- Cactus [13] is an open-source modular environment, which enables parallel computation across different architectures. Modules in Cactus are called “thorns”. A thorn encapsulates all user-defined code. The 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, the 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.

- Charm++ [14] 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. And 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, OpenCL etc. But using these libraries anyway involves the manual management of all the aspects of accelerators programming.
Another possibility to utilize GPU 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. The user should control all the aspects of GPU programming. With a help of GPU Manager, the user can save on some function calls. Still, he should fully control the workflow.

- Chapel [15] is a parallel programming language. Chapel provides the 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 multicore 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 the 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, load balancing, device's execution calls etc, 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.

- Flash Code [16], [17] 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, a lot of 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 cannot 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; etc.

- Some other frameworks and languages. AMUSE [18] 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. Conversely, the utilization of any hardware accelerators should be a part of libraries coupled in a particular configuration.
  Swarm [19] 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 the 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 [20] 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.
  Among other languages, which are not so widely used, we should mention: Julia [21] language, X10 [22]

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 [23], RooFit [24], MLFit [25].

III. ACAF Design and Structure

A. 3 Concepts Design

The design of the ACAF should be both user-friendly for astrophysicists and easily extendable for computer scientists. Therefore, we have designed the ACAF basing on 3 concepts (see Figure 1).

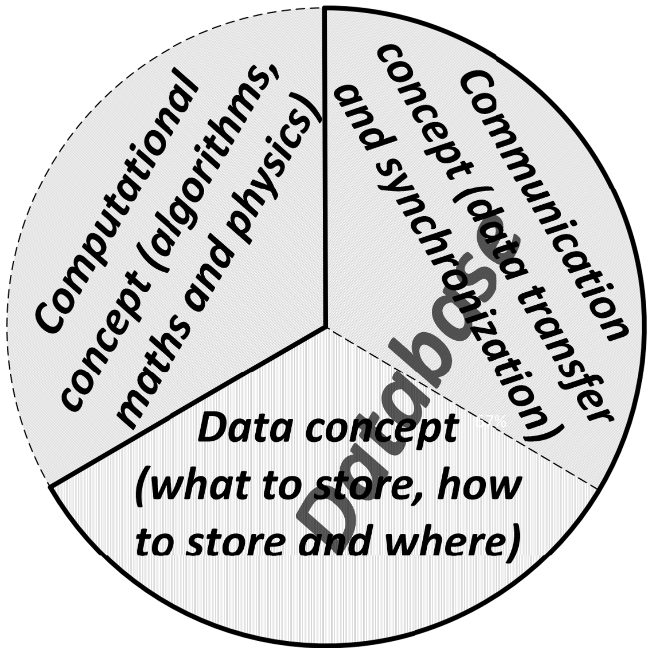
Computational concept (algorithms, maths and physics)
Communication concept (data transfer and synchronization)
Data concept (what to store, how to store and where)
Database

Figure 1. The 3-concepts design.

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 that each algorithm had an efficient implementation for each architecture and device in use. And 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 fulfill the following requirements:

- operating with a set of structures efficient for representing astrophysical data: tuples, trees (oct-trees, k-d trees), arrays;
- operating with huge amount of data;
- the native support of hardware accelerators like 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: the 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 function, device and platform requirements.

This means that this special database can be seen as a partitioned global address space (PGAS), which is already addressed in several existing solutions like Chapel and X10. But in our approach, we incorporate into the database not only partitioning of the address space, also other properties specified above.

Hence in this work, we address only 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.

B. Database Design

The target data for the ACAF database is a set of states of some particle system. According to the definition of a particle system (see Section I), 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, only these types of data are 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 targeted to store only arrays of integral data elements.

As soon as a particle system usually includes some millions of 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 simulations 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., 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 a part of both compiler implementation and the operating system scheduler. At the same time, the programmer's task is to ensure the presence of data in the nearest high-level memory space (usually device Random-access memory (RAM)). Moreover, it is necessary to store data in high-level memory spaces in the format acceptable with computational algorithms. Hence, raw arrays are preserved 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 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. At the same time, 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 technologies we have used include:

- 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. And the full block diagram is shown in Figure 2.

a) Configuration: One of the input data the 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. Particularly, 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.

b) 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.

c) 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. At the same time, the network node sizes are automatically calculated and broadcasted 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.

d) Storage Objects: On the other hand, the context as well defines the storage objects - the instances of the storage interface, which 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.

e) Input and Output Data Definition: Another input the 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.

f) Content Objects: Hence, the data description defines the content objects - the logic how to work with memory. Particularly, a content object determines the following properties of the data:

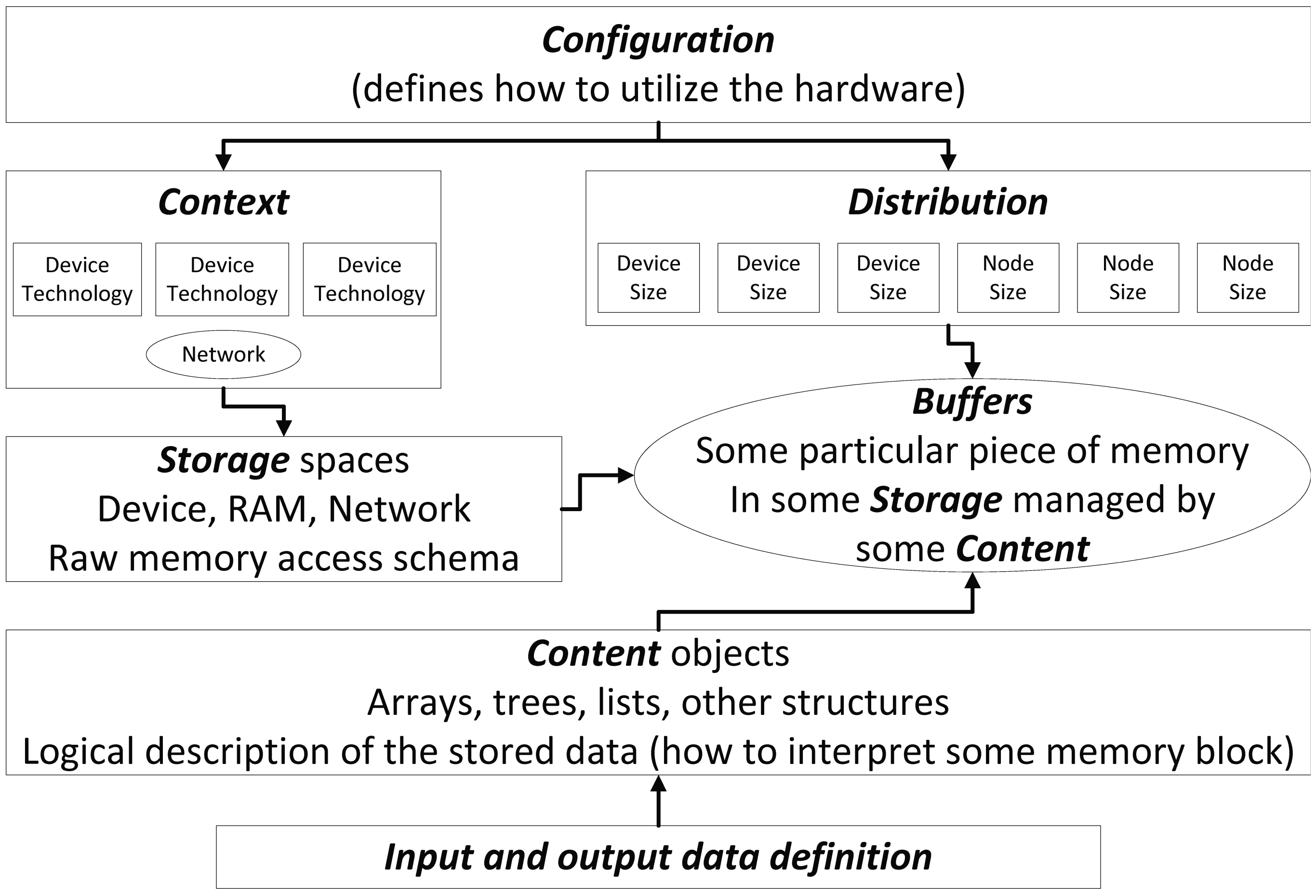

Figure 2. Diagram of ACAF database design.

- 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.

g) 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. At the same time, a memory buffer object itself only contains the reference to the memory region, the size of this region and the storage object, which manages this region.

C. 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. 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.

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.

This means that the computational concept alone requires good programming skills from the astrophysicists, because the precision of the simulation depends on the particular implementation of the mathematical algorithm. 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 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 3 together with the elements of the database design (see Section III-B). Such design includes all the elements necessary to run an astrophysical simulation on a heterogeneous cluster.

a) 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.

b) Implementations: The computational algorithm together with the context object 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.

IV. Implementation Design

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 3) into the implementation design using the UML diagram. The detailed description of the classes including some implementation details is provided in Section V. This section gives the overview of the key mechanisms and techniques used in the implementation described in the following subsections.

A. Device Detection Mechanism

As it was described in Subsection III-B 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 big 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 the 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 etc.) 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 CPU in the system. Since we have targeted the current implementation to work with 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 CPU in the system. This class scans the whole PCI bus of the system in order to find the devices of the VGA type, which are in fact 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.) And then 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).

B. 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 simplify

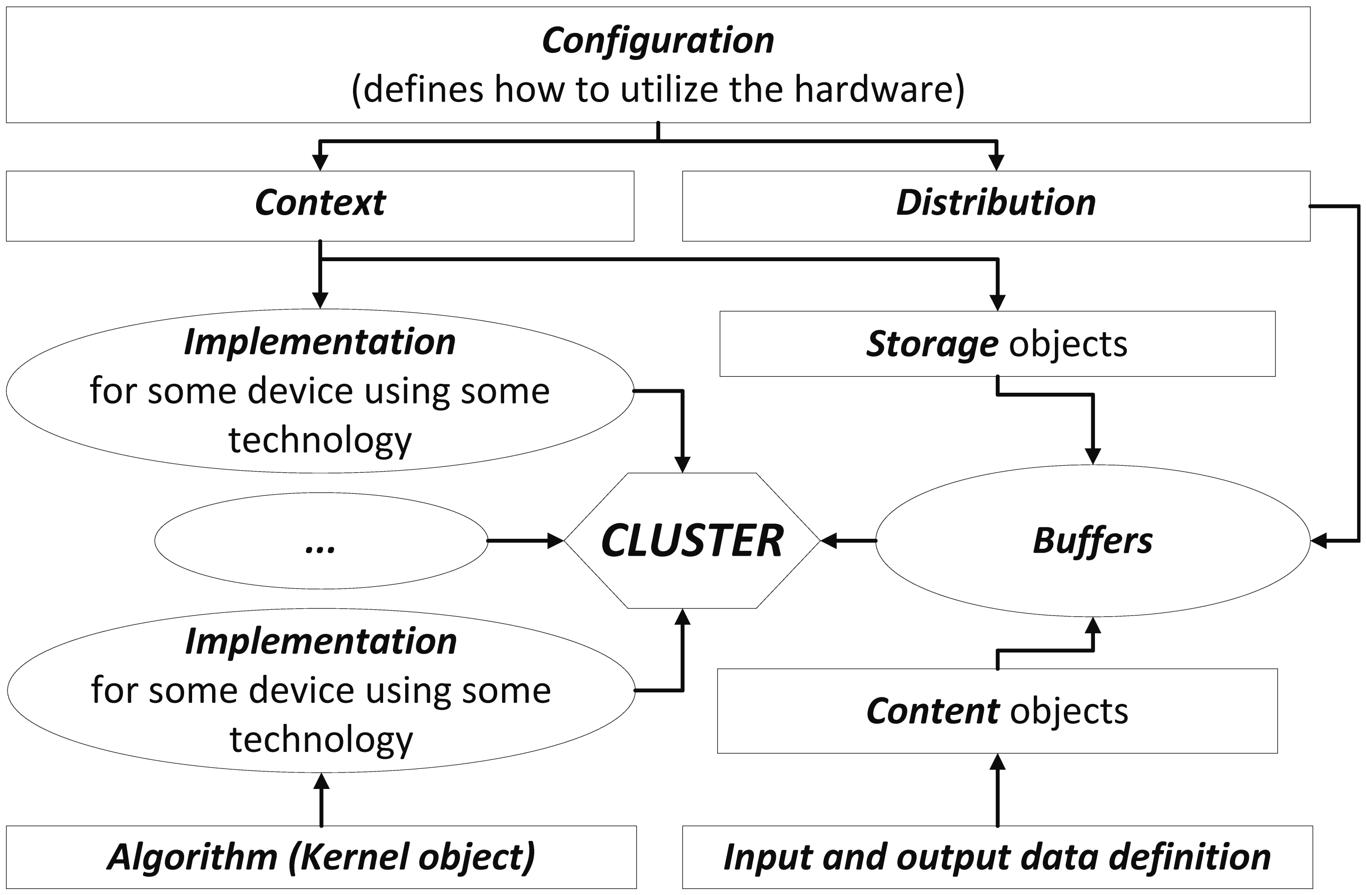
Configuration
(defines how to utilize the hardware)
Context
Distribution
Implementation
for some device using some technology
Storage objects
...
CLUSTER
Buffers
Implementation
for some device using some technology
Content objects
Algorithm (Kernel object)
Input and output data definition

Figure 3. Diagram of the Astrophysical-oriented Computational multi-Architectural Framework design.

the specification of the device-technology association by requesting the user to provide the association using the following possibilities (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 some *Architecture* subclass);
2) the architecture name and the technology name or the keyword "none".

In order to simplify the usage and the implementation of the configuration file specification it was decided to use the libconfig[26] format for the file. The file has the following structure:

- the top-level section "context", which includes the device-technology association specified above, including the optional parameter "skip", which indicates if the unsuccessfully associated devices are to be skipped;
- the top-level optional parameter "network", which specifies the network interface to be used;
- the top-level section "distribution", which contains one or several named subsections;
- each named distribution subsection consists of device-specific blocks: the full device name or the architecture name, size and block vectors.

C. 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 devices 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. And 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.

D. Content Objects And Buffers Instantiation

As it was described in Subsection III-B, 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 - one-dimensional array, multi-dimensional array, oct-tree and so on;
- 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.

E. The Definition of The Computational Concept

As it was described in Subsection III-C, 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. And for each specified content object, the buffer allocated in the device memory is used.

F. 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[26] syntax;
2) create the necessary content objects for all non-scalar distributed algorithm parameters;
3) write technology-specific execution code for all the technologies in use;
4) create the necessary kernel objects;
5) write the main execution logic using content objects and kernel objects in C++ language.

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.

G. 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 Section I. 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 [27].

- The design of the framework preserves the separation of the computational resources by their type (*Architecture* subclasses) and the utilization schema (*Technology* subclasses). This separation forces the 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 the 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.

V. Classes Description

The suggested database is implemented as a part of the framework - the ACAF. The implementation is done in C++ language and is organized as a collection of classes. Some of them are template classes. We concentrate on the key classes used in the ACAF in this section.

A. 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.

B. 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 etc.). 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*.

C. 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 fulfill 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*.

D. 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 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 *NetworkStorage* subclass and adds it to the database.

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

E. 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.

F. 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.

G. 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.

H. 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 V-F);

- a map of the named *Content* objects, where each object represents some data layout schema used in the user application (see Subsection V-G).

And 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.

I. 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. And 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.

J. Extending the ACAF

The user has an opportunity to extend the functionality of the ACAF by implementing the other ancestor classes of the following entities:

- *Architecture* - to support other device types;
- *Technology* - to support other programming technologies;
- *Network* - to support other network protocols;
- *Content* - to support other logical data organizations.

VI. Usage Example

A running example of ACAF usage is represented with several parts: the configuration, the mathematical algorithm implementation and the environmental host code. The provided example represents the code necessary for running distributed NBody simulation on a cluster using MPI for network communication, pthread technology for CPU code and OpenCL technology for GPU code. Any changes in the resource utilization can be made by modifying the configuration file without any need to recompile the program.

A. Configuration File

A configuration file contains the network protocol, the *context* specification and possible distribution descriptions (see Figure 4).

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

Figure 4. The configuration example.

As it was described in Subsection III-B, the example configuration file provides the framework with the hardware utilization schema. Particularly, line-by-line the example file defines the following:

1) The “network” parameter 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 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 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.
    b) Each CPU device processes 256 items per iteration, calculating 4 items per thread job.

B. Algorithm Code (OpenCL and pthread)

According to the technologies specified in the configuration file and the host code initialization routine, the mathematical algorithm should be implemented for one or several technologies. In our example, the algorithm is implemented for OpenCL (see Figure 5) and pthread (see Figure 6) technologies, using respectively OpenCL C language and C++ language. The code of OpenCL implementation is represented as a separate file, while pthread implementation code is a part of the environmental host code and passed to ACAF as a pointer to the function.

```
#pragma OPENCL EXTENSION cl_khr_fp64 : enable
#pragma OPENCL EXTENSION cl_amd_fp64 : enable

#define SOFTENING 0.001

__kernel void force ( uint4 acaf_total,
  __global double  * mass, __global double4 * position,
  __global double4 * velocity, double time_step )
{
  __local double4 shared_position[ITEMS_PER_GROUP];
  size_t lid = get_local_id(0);

  shared_position[lid] = position[get_global_id(0)];
  double4 this_acc;
  this_acc.x = this_acc.y = this_acc.z = this_acc.w = .0;
  for ( size_t i = 0; i < acaf_total.x; ++i )
  {
    double4 dist = shared_position[lid] - position[i];
    this_acc += mass[i] * dist / powr(length(dist) + SOFTENING, 3.);
  }

  size_t gid = get_global_id(0);
  velocity[gid] += this_acc * time_step;
  position[gid] += velocity[gid] * time_step;
}
```

Figure 5. The OpenCL algorithm example.

```
#define SOFTENING 0.001

status force (
  const acaf::uint4 & jid, const acaf::uint4 & jtotal,
  const acaf::variant_vector & args
)
{
  double * mass = reinterpret_cast<double *>(*(args[0].get<void *>()));
  double4 * position = reinterpret_cast<double4 *>(*(args[1].get<void *>()));
  double4 * velocity = reinterpret_cast<double4 *>(*(args[2].get<void *>()));
  double time_step = *(args[3].get<double>());

  double4 this_pos = position[jid[0]];
  double4 this_acc (0.);
  for (size_t i = 0; i < jtotal[0]; ++i)
  {
    double4 dist = this_pos - position[i];
    this_acc += mass[i] * dist / pow(dist.length() + SOFTENING, 3.);
  }

  velocity[jid[0]] += this_acc * time_step;
  position[jid[0]] += velocity[jid[0]] * time_step;

  return error::Success;
}
```

Figure 6. The pthread algorithm example.

```
int main(int argc, char ** argv)
{
  MPI_Init(&argc, &argv);
  status s = error::Success;
  do
  {
    s = acaf::initialize(argc, argv);
    if (s.fail()) break;

    Handle < DataBase > db = Context::getContext()->getDB();
    LinearParticles distr(Context::getContext(), acaf_string("default"));
    Handle<Content> mass, pos, velo;

    {
      acaf::pair<Handle<Content>, status> tmp;
      tmp = db->create< SyncedArray<double, 1> >("mass", distr.units(1));
      if (tmp.second.fail()) cout << tmp.second;
      mass = tmp.first;
      mass->fill(acaf::variant(1.));
      tmp = db->create< SyncedArray<double4, 1> >("position", distr.units(1));
      if (tmp.second.fail()) cout << tmp.second;
      pos = tmp.first;
      pos->random(
        acaf::variant(double4({-1., -1., -1., 0.})),
        acaf::variant(double4({2., 2., 2., 0.}))
      );
      tmp = db->create< LocalArray<double4, 1> >("velocity", distr.units(1));
      if (tmp.second.fail()) cout << tmp.second;
      velo = tmp.first;
      velo->fill(acaf::variant(double4(0.)));
    }

    Kernel force("force", Context::getContext());

    double current_time = 0.;
    double end_time = 1.;
    double time_step = 0.01;

    s = force.add("OpenCL", "gravity.cl", "-cl-mad-enable", true);
    if (s.fail()) cout << s << endl;
    s = force.add("pthread", &::force);
    if (s.fail()) cout << s << endl;
    s = force.set(0, mass);
    if (s.fail()) cout << "Adding mass failed:" << s << endl;
    s = force.set(1, "position");
    if (s.fail()) cout << "Adding position failed:" << s << endl;
    force.set(2, "velocity");
    if (s.fail()) cout << "Adding velocity failed:" << s << endl;
    force.set(3, variant(time_step));
    if (s.fail()) cout << "Adding tinestep failed:" << s << endl;

    while (current_time < end_time)
    {
      fos << "Current time: " << current_time << std::endl;
      s = force.start(distr.units(1));
      if (s.fail()) break;
      pos->synchronize();

      current_time += time_step;
    }
  } while (false);

  acaf::finalize();
  MPI_Finalize();

  if (s.fail())
    printf("An error %d (%s) occurred. Failed!\n", s.code(), s.name());
  else
    printf("Success!\n");

  return s.code();
}
```

Figure 7. The main function example.

C. Environmental Host Code

Finally, the environmental host code represents the main function with initialization instructions, content creations, kernel instantiations and kernel running calls written in C++ programming language with the usage of the classes described in Section IV (see Figure 7).

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*. 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; the contents creation 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 $(0,0,0)$); the kernel creation and adding the available implementations and arguments.
- The particle system evolving loop consists of synchronous execution of the kernel, synchronizing the positions and proceeding to the next time frame.
- Finally, the clean up of the environment also includes 2 function calls symmetric to the initialization: *MPI_Finalize* and *acaf::finalize*.

VII. 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 8, 9 and 10.

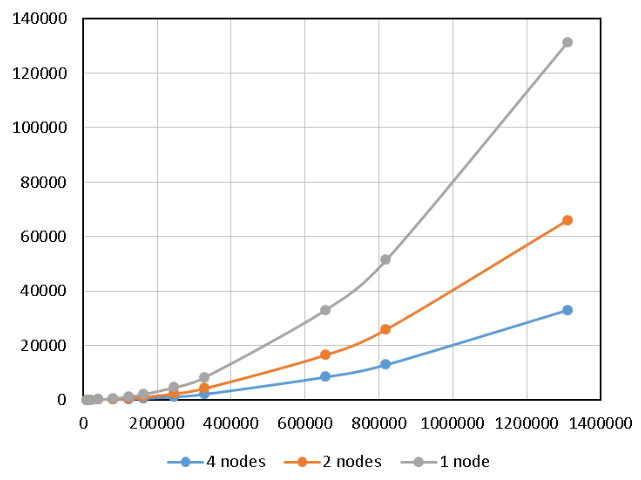

Figure 8. The full comparison chart of running the code on a different number of nodes.

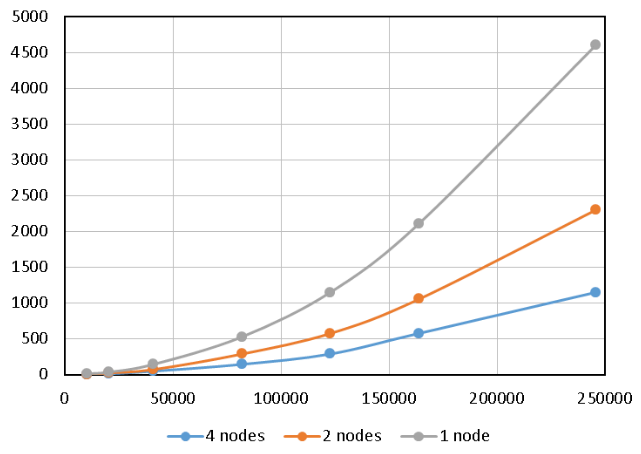

Figure 9. The lower range comparison chart of running the code on a different number of nodes.

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 GPU 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 in the design and implemented in the framework. The average ratios mentioned above consider only the distribution-effective

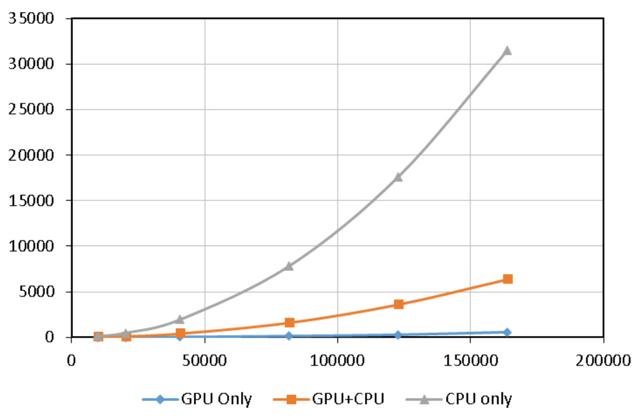

Figure 10. The comparison chart of running the code with different hardware configurations.

execution times, particularly, 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. This comparison shows what is the overhead of using the framework. The bare simulation code consists of the network-distributed computations performed on GPU using the same OpenCL kernel. Figure 11 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.

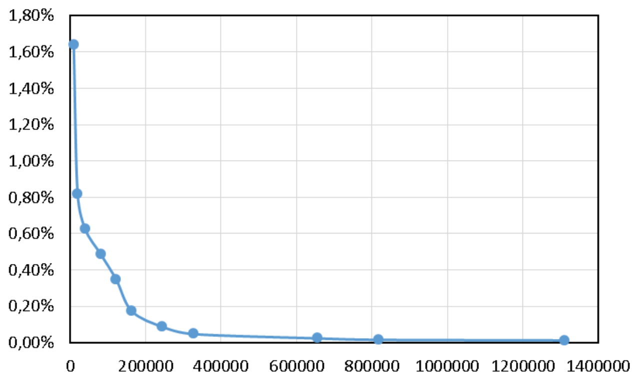

Figure 11. The comparison chart of the ACAF-based implementation to the bare implementation.

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

The tests were carried out on the following test platform: 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 Linux OS. For each test the calculation was equally distributed over all 4 processing nodes.

VIII. Conclusion and Future Work

In this paper, we presented the design details, the implementation aspects and some benchmarking results for the **A**strophysical-oriented **C**omputational multi-**A**rchitectural **F**ramework. The ACAF is targeted to simplify the software development for astrophysical simulations implementation by providing the user with the set of objects and functions covering some aspects of application developing.

In the current work, we focused on the communication and the data concepts of software development problem designing the special distributed database. The database is aimed to process particle systems with float and/or double (integral) parameters. The database aims to store data in high-level memory spaces in the format acceptable with computational algorithms.

The current database implementation utilizes pthreads, OpenCL and CUDA technologies to run the calculation on CPU and GPU devices and MPI interface to distribute and exchange data over the network. The implementation uses 2 types of content: local array and synced array. Extending of the database functionality can be easily done by implementing the certain program interfaces.

We can conclude that the current ACAF implementation facilitates the development of network-enabled heterogeneous NBody force simulation program. With the help of ACAF, the user is able to write an application without the expertise neither in the network programming nor in the parallel programming of some devices (CPU, GPU). ACAF requires the user to do the following tasks:

- Write the configuration file, which specifies the devices and nodes to be used and defines the distribution of the data.
- Implement the mathematical, physical part of the program.
- Write the environmental code, which does the initialization, data definition, data initialization, kernel instantiation and defines the main particle system evaluation loop.

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 II-B 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*, *Content*. This enables a possibility to encapsulate the necessary complex data operations: distribution of the data, its transferring, its synchronizing and etc.
2) The data operations are also separated logically according to the type of the operation: data allocating and transferring is managed by *Storage* classes, data interpretation and logic operations on the data are managed by *Content* classes, while the *Database* class guarantees the correct functionality and provides some misc functions. Such splitting helps to extend only the necessary framework parts.
3) The framework also splits the simulation implementation into several logical 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;
    - the kernel implementations represent mathematical and physical parts of the code;
    - the environmental code does the initialization, data definition, data initialization, kernel instantiation and defines the main particle system evaluation loop.
4) The framework is designed as a C++ framework. This means that the user and the framework developer have an access to a big range of different powerful system calls and a variety of computational libraries and tools. So, the user has a choice either to reimplement 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. The encapsulation of the operations implies that the final user should not know and use some device-specific functions, libraries and tools. The framework classes separate also the functional aspects of the work with some device: *Architecture* class enables the devices of some specific type to be recognized and used by the framework; *Technology* class focuses on the device utilization schema. This separation facilitates the extension possibilities of the framework: the developer is able to target one of the aspects.
6) The framework also encapsulates the network-distributed communications and computations. This encapsulation enables the final users to avoid the network-related operations and to switch easily between the single-node configuration and the multi-nodes configuration.

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, like OpenCL C and CUDA C for utilizing GPU. The computational kernel used by the framework to execute the actual calculations, should be implemented by the user for each technology combined in the current computational context. To avoid the necessity of having the individual kernel implementations for each technology, it is mandatory to design and implement some common parallel programming language, which can be further translated into the technology-specific languages. Still, it is crucial to preserve an ability to

use the native technology-specific programming languages in order to be able to finely tune a particular computational code.

2) Additionally to the technology-targeted programming languages requirement, the reentrance of the user-defined computational code is necessary, which 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.
3) In comparison to the other approaches described in Subsection II-B, the proposed framework design still requires a lot of coding work to be done. The amount of coding can be reduced by implementing the Domain-Specific Language as a layer over the framework functionality.
4) The framework targets exclusively 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.
5) There is no possibility to provide immediately some user-driven testing of the framework, since the main advantage of many other approaches (like Flash Code framework, see Subsection II-B) is 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 the most of the 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.

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 NBody 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 big 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 big 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 also encapsulates the dynamic function semantic additionally 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 cannot 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 big data on the cluster with poor local storage capacities, when the full range of all the necessary data cannot 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), etc. 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 preserve an ability to switch to the direct framework calls and to provide the kernel implementation in the technology-native programming languages.

References

[1] D. Razmyslovich, G. Marcus, and R. Männer, “Towards an astrophysical-oriented computational multi-architectural framework,” in COMPUTATION TOOLS 2016 : The Seventh International Conference on Computational Logics, Algebras, Programming, Tools, and Benchmarking. IARIA, 2016, pp. 19 – 26.

[2] N. Nakasato, G. Ogiya, Y. Miki, M. Mori, and K. Nomoto, “Astrophysical Particle Simulations on Heterogeneous CPU-GPU Systems,” Jun. 2012. [Online]. Available: http://arxiv.org/abs/1206.1199

[3] R. Spurzem et al., “Astrophysical particle simulations with large custom GPU clusters on three continents,” Computer Science - Research and Development, vol. 26, no. 3-4, Apr. 2011, pp. 145–151. [Online]. Available: http://www.springerlink.com/index/10.1007/s00450-011-0173-1

[4] 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, no. November. IEEE, Nov. 2010, pp. 1–9.

[5] S. Braibant, G. Giacomelli, and M. Spurio, Particles and fundamental interactions: an introduction to particle physics, 2nd ed. Springer, 2011.

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

[7] 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, vol. 26, no. 3-4, 2011, pp. 257–266. [Online]. Available: http://dx.doi.org/10.1007/s00450-011-0171-3

[8] L. Dagum and R. Menon, “Openmp: An industry-standard api for shared-memory programming,” IEEE Comput. Sci. Eng., vol. 5, no. 1, Jan. 1998, pp. 46–55. [Online]. Available: http://dx.doi.org/10.1109/99.660313

[9] Khronos Group - OpenCL. [Online]. Available: http://www.khronos.org/opencl [retrieved: Nov., 2016]

[10] CLara. [Online]. Available: https://www.alpha-tierchen.de/ bkoenig/clara/ [retrieved: Nov., 2016]

[11] Khronos Group - SyCL. [Online]. Available: http://www.khronos.org/sycl [retrieved: Nov., 2016]

[12] A. Dubey et al., “A survey of high level frameworks in block-structured adaptive mesh refinement packages,” Journal of Parallel and Distributed Computing, 2014.

[13] T. Goodale et al., “The Cactus framework and toolkit: Design and applications,” in Vector and Parallel Processing – VECPAR’2002, 5th International Conference, Lecture Notes in Computer Science. Berlin: Springer, 2003. [Online]. Available: http://edoc.mpg.de/3341

[14] P. Jetley, L. Wesolowski, F. Gioachin, L. V. Kalé, and T. R. Quinn, “Scaling hierarchical n-body simulations on gpu clusters.” in SC. IEEE, 2010, pp. 1–11. [Online]. Available: http://dblp.uni-trier.de/db/conf/sc/sc2010.html

[15] B. Chamberlain, “Chapel (cray inc. hpcs language).” in Encyclopedia of Parallel Computing, D. A. Padua, Ed. Springer, 2011, pp. 249–256. [Online]. Available: http://dblp.uni-trier.de/db/reference/parallel/parallel2011.html

[16] A. Dubey et al., “The software development process of flash, a multiphysics simulation code.” in SE-CSE@ICSE, J. Carver, Ed. IEEE, 2013, pp. 1–8. [Online]. Available: http://dblp.uni-trier.de/db/conf/icse/secse2013.html

[17] B. Fryxell et al., “FLASH: An adaptive mesh hydrodynamics code for modeling astrophysical thermonuclear flashes,” Astrophys. J. Supp., vol. 131, Nov. 2000, pp. 273–334. [Online]. Available: http://dx.doi.org/10.1086/317361

[18] S. Zwart, “The astronomical multipurpose software environment and the ecology of star clusters.” in CCGRID. IEEE Computer Society, 2013, p. 202. [Online]. Available: http://dblp.uni-trier.de/db/conf/ccgrid/ccgrid2013.html

[19] S. Dindar et al., “Swarm-ng: a cuda library for parallel n-body integrations with focus on simulations of planetary systems,” CoRR, vol. abs/1208.1157, 2012. [Online]. Available: http://dblp.uni-trier.de/db/journals/corr/corr1208.html

[20] The Enzo project. [Online]. Available: http://enzo-project.org/ [retrieved: Nov., 2016]

[21] J. Bezanson, A. Edelman, S. Karpinski, and V. B. Shah, “Julia: A fresh approach to numerical computing,” CoRR, vol. abs/1411.1607, 2014. [Online]. Available: http://arxiv.org/abs/1411.1607

[22] P. Charles et al., “X10: An object-oriented approach to non-uniform cluster computing,” SIGPLAN Not., vol. 40, no. 10, Oct. 2005, pp. 519–538. [Online]. Available: http://doi.acm.org/10.1145/1103845.1094852

[23] C. Feichtinger, S. Donath, H. Köstler, J. Götz, and U. Rüde, “WaLBerla: HPC software design for computational engineering simulations,” Journal of Computational Science, vol. 2, no. 2, May 2011, pp. 105–112. [Online]. Available: http://dx.doi.org/10.1016/j.jocs.2011.01.004

[24] I. Antcheva et al., “{ROOT} - a c++ framework for petabyte data storage, statistical analysis and visualization,” Computer Physics Communications, vol. 180, no. 12, 2009, pp. 2499 – 2512, 40 {YEARS} {OF} CPC: A celebratory issue focused on quality software for high performance, grid and novel computing architectures. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0010465509002550

[25] A. Lazzaro, S. Jarp, J. Leduc, A. Nowak, and L. Valsan, “Report on the parallelization of the MLfit benchmark using OpenMP and MPI,” CERN, Geneva, Tech. Rep. CERN-OPEN-2014-030, Jul 2012. [Online]. Available: https://cds.cern.ch/record/1696947

[26] libconfig – C/C++ configuration file library. [Online]. Available: http://www.hyperrealm.com/libconfig/ [retrieved: Nov., 2016]

[27] TOP500. [Online]. Available: https://en.wikipedia.org/wiki/TOP500 [retrieved: Nov., 2016]