Towards an Astrophysical-oriented Computational multi-Architectural Framework

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

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

***Abstract*—In this exploratory paper, we present a framework for simplifying software development in the astrophysical simulations branch - Astrophysical-oriented Computational multi-Architectural Framework (ACAF). The ACAF is designed to provide a user with the set of objects and functions covering some aspects of application development for astrophysical problems. The target data to be processed with the ACAF is a set of states of a particle system. Being designed as a C++ framework, the ACAF decreases the expertise needs required to implement such programs preserving the extension flexibility and the possibility to use the existing libraries. The ACAF abstracts the accelerating device itself, the usage of it, the data distribution and usage. Also, the ACAF incorporates the different kernel implementations into a single object.**

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

I. Introduction

Astrophysical simulation tasks have usually high computational density, therefore it's common to use hardware accelerators for solving them [1]. 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 [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. While the computational tasks embrace numerical solving of a number of equations, which evaluate the state of a particle system [5].

Developing astrophysical simulation applications for heterogeneous clusters without usage of specialized frameworks and libraries requires the following knowledge:

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

This means much time and expertise for astrophysicists, what restricts the scientists to perform calculation experiments easily on clusters 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. In this exploratory paper, we present the current state of art and the results of some experiments with the ACAF.

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 a 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 programming aspects mentioned above;
- in the same time, there should be a possibility to extend the tools in use as well as to provide the alternative implementations of existing tools;
- finally, it would be an additional advantage to preserve a possibility to reuse the existing computational libraries, when it makes sense.

The rest of the paper is divided into 4 sections. The Section II highlights the currently existing standards, frameworks and languages for the software development targeted heterogeneous systems. The Section III consists of 3 subsections, each of them presenting some design motivations and solutions we have used to reach the goal. In the Section IV, the usage example of the current framework implementation is given. Finally, the Section V 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

- OpenMP [6] is a standard Application Programming Interface (API) for shared-memory programming, which enables easy and efficient CPU utilization on a single node using compiler directives. The latest version 4.0 includes the compiler directives for hardware accelerators, which were previously presented in another branch OpenACC [7]. OpenMP API model definitely reduces the requirements in parallel programming skills abstracting the numerous function calls in easy-readable pragmas. Still, it doesn't hide a lot of implementation details, which are out-of-interests for scientific programmers: device data allocation, data transferring, runtime synchronization, etc. Also, API doesn't cover at all any kind of network communications and is designed solely for a single node.
- Message Passing Interface (MPI) [8] 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. Using the extension MVAPICH2 [9], it is also possible to integrate CUDA-enabled GPU data movement transparently into MPI calls.
- CUDA [10] stands for Compute Unified Device Architecture and is a parallel computing platform and API model created by Nvidia and is only used for Nvidia GPUs. CUDA enables a user to utilize Nvidia GPUs for general-purpose computations on a single node.
- OpenCL [11] is an open standard for general purpose parallel programming across different heterogeneous processing platforms: CPUs, GPUs and others. Choosing OpenCL, a user can utilize some hardware accelerators on a single node.
- SyCL [11] is a new C++ single-source heterogeneous programming model for OpenCL. SyCL benefits from C++11 features like lambda functions and templates. SyCL provides a high level programming abstraction for OpenCL 1.2.

B. Frameworks and Languages

- Cactus [12] is an open-source modular environment, which enables parallel computation across different architectures due to its modularity. As separate modules, Cactus code also provides CUDA and MPI utilization. Additionally, there is an extension of Cactus code - CaCUDA, which is able to utilize Nvidia GPUs across cluster nodes by converting CaCUDA source code into CUDA kernels. No other hardware accelerators are supported so far. As of 2015, CaCUDA looks like to be not developed any more.
- Charm++ [13] 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. Charm++ library is responsible for distributing chares between the processing units and establishes the communication between them. Charm++ provides also an additional library - Charm++ GPU Manager, which enables the user to utilize GPU directly from Charm++. In order to run some code on GPU, a user should define a work request for GPU Manager providing CUDA kernel, input and output arguments to be transferred to GPU. GPU Manager ensures the overlapping of transfers and executions on GPU and runs GPU kernel asynchronously.
- Chapel [14] is a parallel programming language. Chapel provides a user with a high-level parallel programming model which supports data parallelism, task parallelism and nested parallelism. Chapel is a very powerful language, which enables the user to write the parallel programs with several lines of code. While being designed as a new standalone language, Chapel has limited possibilities for extending the functionality - since Chapel is an open source project, everybody can have the source code and change Low Level Virtual Machine (LLVM) grammar for having new commands. But this means, even reusing the existing computational libraries could be only done with new language features.
- Flash Code [15][16] is a modular Fortran90 framework, which uses MPI to distribute the calculations over the cluster nodes. The Flash system has no built-in support for any hardware accelerators and relies on the particular modules to optimize the calculations as much as possible. A module in the Flash system is some atomic algorithmic routine performing mathematical evaluation of the particle system. This means that a module can be implemented using any accelerating techniques and libraries, but anyway requires the proficiency in parallel programming (hardware accelerators utilization; MPI usage; data distribution and

synchronization using MPI and hardware accelerators, etc). The Flash system is deployed with a big number of modules.

- Swarm [17] is a CUDA library for parallel n-body integrations with focus on simulations of planetary systems. The Swarm framework targets single machines with Nvidia GPUs as hardware accelerators. The framework provides a user with a possibility to extend the calculations algorithm, but the final system isn't scalable and cannot utilize the power of a cluster. It is designed to solve some specific problems.
- AMUSE [18] is 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. While the utilization of any hardware accelerators should be a part of libraries coupled in a particular configuration.
- The Enzo [19] project is a community-developed adaptive mesh refinement simulation code. The code is modular and can be extended by users. Enzo doesn't support network communication. Still, it contains several modules developed to utilize Nvidia GPUs using CUDA.
- Some other languages, which aren't that widely used: Julia [20] language, X10 [21] language, Fortress language [22]. All these languages were initially designed for CPU clusters. Some of them provide ports or extensions for hardware accelerators, which usually have no abstraction for the accelerator memory space communications.
- And other widely used domain-specific libraries: WaLBerla [23], RooFit [24], MLFit [25].

III. ACAF Design and Structure

The design of the ACAF should be both user-friendly for astrophysicists and easily extendable for computer scientists. Therefore, we've designed the ACAF basing on 3 concepts:

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 function in the space of user tools has an efficient implementation for each architecture and device in use. And all implementations for the same function can work together on different platforms.
2) The **communication concept** describes data transfers and synchronization points between computing units or storages. The concept lies in efficient data-distribution mechanisms, which warranty 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 transforming it according to the user-defined, architecture-defined or device-defined rules.
3) The **data concept** describes logical and physical representation of the data used in a solution, as well as distribution of this data between different storages. 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.

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.

While 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: a user should be able to extend the number of features in use - architectures; devices; data-structures; data manipulation schemes and functions; communication protocols;
- the database should store the data according to the 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.

A. 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, these types of data are only considered. A state of some particle system can be represented in some computer memory space as an array of structures, where members of a structure are particle parameters, e.g., integral data types. Therefore, the ACAF database is only targeted to store arrays of integral data elements.

As soon as a particle system usually includes some millions of particles, it's 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 (processing registers and near by caches of a unit) is a part of both compiler implementation and the operating system scheduler. While 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's 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 implementation of the computational concept particular to a certain problem. While data synchronization in heterogeneous computer clusters context 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've used include: threads and OpenCL 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 to another; synchronizing the memory buffer content distributed between different technologies.

B. Database Implementation

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. The current framework implementation is targeted to be built and executed on machines running Unix operating system.

1) Device: A *device* instance represents some computational device, which can be used for simulation calculations on the current node. A device instance is always described by some *architecture* instance, the vendor name, the vendor identifier, the device name, the device identifier and a set of *technology* instances. The format and type of the identifiers are always architecture-dependent. All device instances are created automatically by ACAF during framework initialization according to the devices found in the operating system. No manual instantiation of a device class is possible.

2) Architecture: Architecture class is an interface class for any *device* type supported by ACAF. The instance of each ancestor architecture type is a singleton in any ACAF process. This instance provides the functionality to identify all computational devices of the desired type in the current cluster node. The predefined architecture ancestor types are:

- CPUArchitecture - identifies all CPU devices presented in the current node by parsing */proc/cpuinfo* file;
- GPUArchitecture - identifies all GPU devices presented in the current node by scanning all Peripheral Component Interconnect (PCI) devices of Video Graphics Array (VGA) type.

3) Technology: Technology class is an interface class for describing some programming technology, which can be used for some devices presented in the current node. The instance of each ancestor technology type is a singleton in any ACAF process. The ancestor class describes how to utilize a device for computational purposes: which devices are supported; which programming language (if any) should be used; how to set the parameters before executing the code; which *storage* class should be used for storing buffers; etc. Assigning the correct set of technology instances for each device is also done automatically during the initialization of ACAF. The predefined technology ancestor types are:

- pthreadTechonology - includes the functionality to run native functions in several threads using Unix pthreads library;
- OpenCLTechnology - includes the functionality to run OpenCL kernels on the supported devices;
- CUDATechnology - includes the functionality to run precompiled CUDA kernels on Nvidia devices.

4) Network: Network class is an interface class for describing some network protocol to utilize network-based computer clusters. The instance of a network type is a singleton in any ACAF process. The ancestor class describes:

- the topology of the current process instances distributed over the network;
- the communication protocol between the processes of different cluster nodes (implemented as a *storage* class type);
- the synchronization mechanisms between nodes.

The selection of the particular ancestor network type is done according to the configuration provided by the user. ACAF predefines only one network ancestor type: MPINetwork which includes the functionality to utilize MPI library.

5) Context: The *context* instance is a set of devices and the programming *technologies* to be utilized for executing the simulation.

6) Database: The *database* instance is a part of *context*, which manages the data used in the scope of the parent context. The database object has a set of *storage* instances and a set of *content* instances. The database instance is the main user interface to manipulate the data: to list all available storages in the system; to create new distributed content instances; to list the existing content instances.

7) Storage: The *storage* class is the interface for allocating/reading/writing/synchronizing data in the associated memory space. The storage classes implemented in ACAF are divided into the following categories:

- RAMStorage serves the functionality to operate with on-board RAM of the current node. This object is a singleton for a database.
- DeviceStorage serves the functionality to operate with some built-in device high-level memory (usually device RAM), like GPU or FPGA. For example, CUDA storage or OpenCL storage are typical device storage instances. Usually, the implementation of a particular device storage type is *technology*-dependent. Therefore, any device storage is instantiated by the *technology* instance used in the current context for the particular device.
- FileStorage serves the functionality to operate with the files, available in the current operating system.
- NetworkStorage serves the functionality to operate with the remote content. The network storage implementation is *network*-dependent. The network storage instances are created automatically by the ACAF during the initialization according to the *network* ancestor class currently used.

Each storage instance has a collection of buffers, which are physical or abstract regions in some memory space. A storage instance doesn't reflect the logical organization of the data and only operates with its representation in the memory (some sequence of bytes).

8) Content: The *content* class is the interface for logical organization of the data stored in the *storage* instances of the database. An instance of the *content* class reflects the particular representation of data in some memory buffer. Additionally, the ancestor content classes provide the user with some data manipulation functions, like initializing, dumping, synchronizing data. The ACAF predefines the following content types:

- Array class is a template class, which represents a distributed array with elements of the template type: each calculation device in the context comprises some part of the array. The parts are completely independent. A typical example of an array is masses of particles.
- SyncedArray class is a template class, which represents a synchronized distributed array with elements of the template type: each calculation device in the context comprises the full array, but owns only some part of it. This means that the device should modify only its own part, while the rest array will be synchronized time-to-time according to the algorithm. A typical example of a synced array is positions of particles.

9) Kernel: A *kernel* represents some atomic function, which is run on the computational devices of the context. Each kernel instance contains a collection of its implementations, where each *kernel implementation* represents some binary-coded *technology*-dependent executable function. The instances of the kernel class are created by the user according to the computational algorithm.

10)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.

IV. 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 1).

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

Figure 1. The configuration example.

The current configuration file consists of 2 sections:

- The first section specifies which devices and nodes should be used for running the calculation (parameters *network* and *context*). Particularly, the example file above specifies that the calculation is going to be distributed over the network with a help of MPI interface and that on each node of the network all CPUs are going to be utilized by pthread technology and all GPUs are going to be utilized by OpenCL technology. An additional flag *skip* identifies that all devices unsupported by the specified technologies should be skipped.
- The second section specifies the distribution of the data inside of a single node. The user can specify as many different distributions as it's necessary using the distinct names. In our example, we have a single distribution with the name *default*.

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 2) and pthread (see Figure 3) 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 (
  __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 < get_global_offset(0) + get_global_size(0); ++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 2. 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 3. The pthread algorithm 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 III-B (see Figure 4).

```
int main(int argc, char ** argv)
{
  MPI_Init(&argc, &argv);
  status s = error::Success;
  do
  {
    // initialize the framework, the configuration file will be read
    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;

    {
      // create arrays for masses, positions and velocities
      acaf::pair<Handle<Content>, status> tmp;
      tmp = db->create< Array<double, 1> >("mass",
                 Content::ACAF_CONTENT_GLOBAL, 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",
                 Content::ACAF_CONTENT_GLOBAL, 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< Array<double4, 1> >("velocity",
                 Content::ACAF_CONTENT_GLOBAL, distr.units(1));
      if (tmp.second.fail()) cout << tmp.second;
      velo = tmp.first;
      velo->fill(acaf::variant(double4(0.)));
    }

    double current_time = 0.;
    double end_time = 1.;
    double time_step = 0.01;
    // instanciate kernel
    Kernel force("force", Context::getContext());
    // add kernel implementations - OpenCL and pthreads
    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;
    // add kernel arguments, which will be forwarder later to
    // the implementations code
    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;
    s = force.set(2, "velocity");
    if (s.fail()) cout << "Adding velocity failed:" << s << endl;
    s = force.set(3, variant(time_step));
    if (s.fail()) cout << "Adding timestep failed:" << s << endl;

    // evaluate the particle system
    while (current_time < end_time)
    {
      // run the kernel
      s = force.start(distr.units(1));
      if (s.fail()) break;
      // synchronize positions all-to-all
      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 4. The main function example.

V. Conclusion and Future Work

In this exploratory paper, we presented the current state of art and some 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 a 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: 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 a 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 some environmental code, which does the initialization, data definition, data initialization, kernel instantiation and defines the main particle system evaluation loop.

We performed the comparison tests of the ACAF-based implementation of the Nbody forces simulation (see Section IV) against the bare OpenCL/MPI implementation. The Figure 5 represents the percent overhead of the execution time of the ACAF-based implementation to the execution time of the bare implementation scaled over the particles number in the example system. According to this chart, we see, that the time overhead of using ACAF drops to less than 1 percent for the bigger particle systems, which equivalents to 97 seconds for the case of 327680 particles.

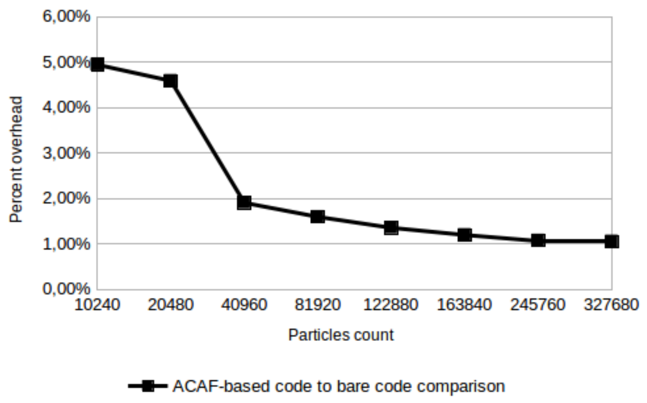

Figure 5. The ACAF-based implementation to bare implementation comparison chart.

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. The calculation was performed only on the GPUs using OpenCL and the positions of the particles were synchronized after each iteration using MPI.

In comparison with the other approaches mentioned in Section II-B, the following advantages of our approach can be mentioned:

1) ACAF is designed as a C++ framework in the first place. This implies that a lot of different other existing libraries and tools can be reused when necessary. So the user has a choice either to reimplement the algorithm using the framework tools or reuse the existing solution.
2) ACAF is designed to be domain specific for astrophysical (particle) problems, therefore it can have lighter structures as the generic tools.
3) ACAF abstracts the device itself and the usage of the device as well. This allows to create one's own usage schemas (*Technologies*) as well as extend the devices supported (by implementing *Architecture* interface).
4) ACAF abstracts the data distribution and usage, while providing still the flexibility for the user to implement other *Storage* classes and other *Content* classes. The *Storage* classes can also represent files, which abstracts input-output operations in the same manner.
5) ACAF incorporates the different kernel implementation into one object. And the object "knows" where and how to execute the code.

While the current implementation of the ACAF has the following limitations:

1) ACAF requires the usage of the extended C languages, like OpenCL and CUDA for utilizing GPUs.
2) The user should be aware that the computational code will be run simultaneously on different data and therefore the code should be reentrant.
3) ACAF provides only arrays as the content objects.

In the future, it's necessary to improve ACAF by extending it with the following features:

- Implement tree-structure content, which can be directly utilized for advanced SPH and NBody simulations.
- Implement the support of astrophysical-native file formats: Hierarchical Data Format version 5 (HDF5), Flexible Image Transport System (FITS), etc.
- Move ACAF implementation forward by introducing the domain specific language, which will eliminate the separate implementations for each technology.
- Implement partially synchronized arrays, enabling so even bigger data ranges.

REFERENCES

[1] R. Spurzem et al., "Accelerating astrophysical particle simulations with programmable hardware (FPGA and GPU)," Computer Science - Research and Development, vol. 23, no. 3-4, May 2009, pp. 231–239. [Online]. Available: http://www.springerlink.com/index/10.1007/s00450-009-0081-9

[2] N. Nakasato, G. Ogiya, Y. Miki, M. Mori, and K. Nomoto. Astrophysical Particle Simulations on Heterogeneous CPU-GPU Systems. [Online]. Available: http://arxiv.org/abs/1206.1199 [retrieved: Feb., 2016]

[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] 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

[7] OpenACC. [Online]. Available: http://www.openacc.org/ [retrieved: Feb., 2016]

[8] 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.

[9] H. Wang et al., “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

[10] Nvidia CUDA. [Online]. Available: http://www.nvidia.com/object/cuda_home_new.html [retrieved: Feb., 2016]

[11] Khronos Group. [Online]. Available: http://www.khronos.org [retrieved: Feb., 2016]

[12] 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, pp. 197–227. [Online]. Available: http://edoc.mpg.de/3341

[13] 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

[14] B. L. 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

[15] 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

[16] 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

[17] 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, pp. 6–18. [Online]. Available: http://dblp.uni-trier.de/db/journals/corr/corr1208.html

[18] S. P. 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] The Enzo project. [Online]. Available: http://enzo-project.org/ [retrieved: Feb., 2016]

[20] J. Bezanson, A. Edelman, S. Karpinski, and V. B. Shah, “Julia: A fresh approach to numerical computing,” CoRR, vol. abs/1411.1607, 2014, p. 517. [Online]. Available: http://arxiv.org/abs/1411.1607

[21] 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

[22] Fortress project. [Online]. Available: http://projectfortress.java.net [retrieved: Feb., 2016]

[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