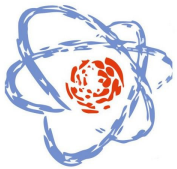
A lightweight library prototype for Monte-Carlo simulation of relativistic nucleus-nucleus collisions based on pipeline architecture

S. Savenkov [1,2]
A. Novikov [1,3]
K. Vasyagina [1]
A. Svetlichnyi [1,2]

1 – Moscow Institute of Physics and Technology
2 – Institute for Nuclear Research of the Russian Academy of Sciences
3 – Yandex



Outline

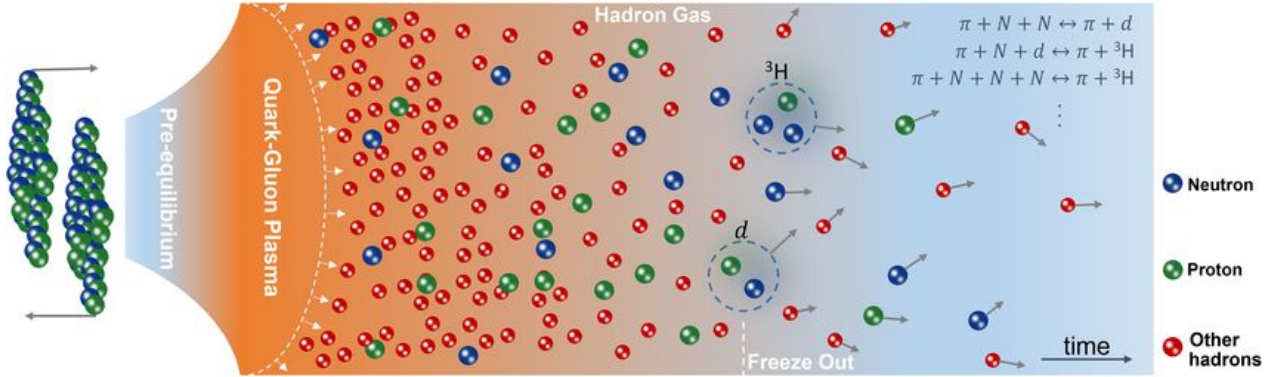
- Similarities between nuclear collision modelling and data pipeline architecture.

- Design of a C++ library for code organization:
  - modules
  - abstract factory
  - build system and integration.

- Examples of the library usage.

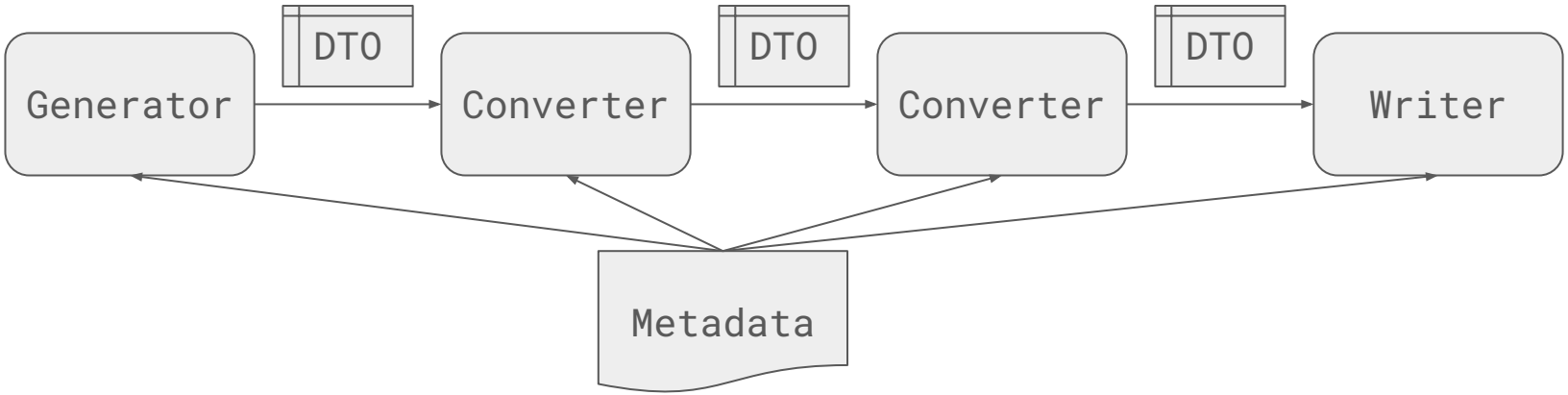


Nucleus-Nucleus collision modelling



- There are a lot of different models and even more implementations: QMD, Glauber model species, PHSD, QGSM, SMM, fragmentation models, plenty of afterburners, etc.
- We need to include several models to describe different physics, however trying to describe everything proves to be too performance demanding.
- Despite similarities between their input and output, combining them requires additional converter scripts or some advanced interoperability and requires a lot of effort.
- Significant amount of modern codes is written in C++ with its limits on interoperability.

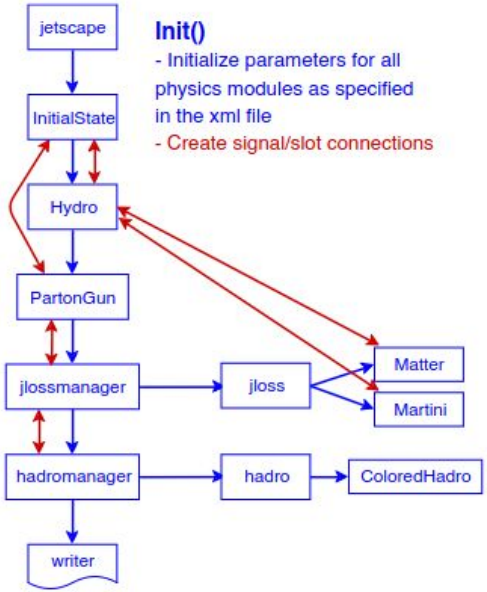
Data Pipeline Architecture



- Every step of data processing is represented by a filter that is independent from other filters.
- Filters are pre-configured by external metadata: data that isn't directly involved in data processing.
- Consistent Data Transfer Object (DTO) between stages.

Inherently similar to the nucleus-nucleus modelling process on a general timescale!

Existing solutions



J.H. Putschke et. al. arXiv:1903.07706

- Specialised data storage formats (HepMC, MCPL, etc.)
  - Possible standardization.
  - Actual integration of different models isn't possible: converter scripts are still required.
  - Most experiments stick to this solution.
- JETSCAPE
  - Focuses on jet modeling.
  - Possible to create new modules.
  - Modularization is at low level – resulting programs can be quite complex.

COllision LAyout for Colliders (COLA)

- C++17 library for code organization without any external dependencies. This standard has been chosen for compliance with modern ROOT6(7) versions.
- Resulting programs contain the whole modeling pipeline in one executable file.
- Engrained modularity: each model is a filter in the data pipeline.
- Filters are provided by separate libraries with their own set of dependencies and are exposed to control code via dependency injection (DI) process.
- Filters are configured by an XML file during modeling pipeline construction via separate factory classes. Therefore, it isn't needed to build the code to change a model.
- CMake integration: COLA library and modules are CMake packages.

https://github.com/Spectator-matter-group-INR-RAS/COLA



COLA Structure

```cpp
/** Particle data.
 *  A structure representing data about a single particle
 */
struct Particle {
    AZ getAZ() const;

    LorentzVector position; /**< Position <t, x, y, z> vector. */

    LorentzVector momentum; /**< Momentum <e, x, y, z> vector. */

    int pdgCode;    /**< PDG code of the particle. */
    ParticleClass pClass;   /**< Data about particle origin. See ParticleClass for more info.*/
};

class VConverter : public VFilter {
public:
    VConverter() = default;
    VConverter (const VConverter&) = delete;
    VConverter (VConverter&&) = delete;
    VConverter& operator=(const VConverter&) = delete;
    VConverter& operator=(VConverter&&) = delete;
    ~VConverter() override = 0;

    /** A method to process one event by the converter model.
     *  Users are supposed to override this method to process a single event by the converter model.
     *  @param data A pointer to the EventData to be processed.
     *  @return A pointer to the EventData of the processed event.
     */
    virtual std::unique_ptr<EventData> operator()(std::unique_ptr<EventData>&& data) = 0;
};
```

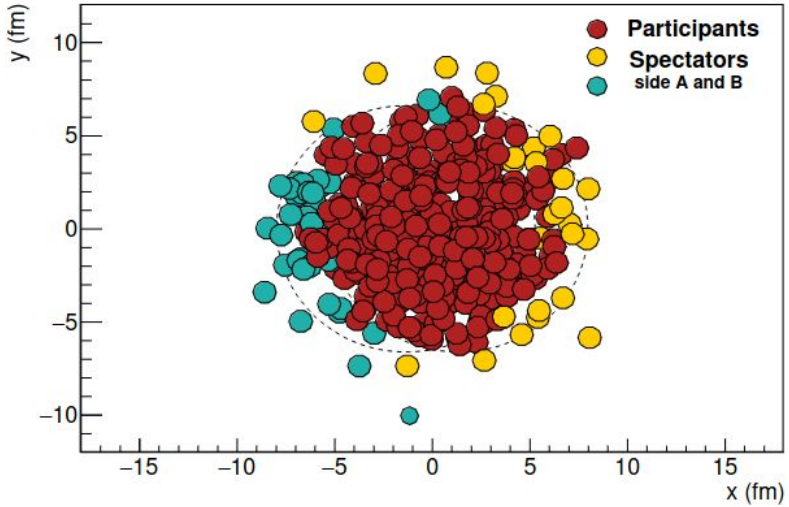
- Data transfer object (DTO), accurate Lorentz vector implementation.
- Abstract filter classes and factory.
- Manager classes: registration of filter factories and config file parsing by TinyXML-2* (DI).
- CMake build and installation instructions.

*) https://github.com/leethomason/tinyxml2/tree/master



CGlauber generator

- An event generator module that is a wrapper of a popular TGlauberMC implementation of the Glauber model. This implementation is used in our AAMCC-MST model.
- Nuclei are set up stochastically and then “collided”, assuming the nucleons are propagating in a straight line along the beam axis, with them being marked as wounded or spectator.



S. Loizides, J. Kamin, D. d’Enterria, Phys. Rev. C, vol. 97, p. 054910, 2018

CGlauber generator

```
class CGlauberGenerator final: public cola::VGenerator{
private:
    int pdgCodeA;
    int pdgCodeB;
    double pZA;
    double pZB;
    double energy;
    double sNN;
    double xSectNN;

    std::unique_ptr<TGlauberMC> generator;
    std::unique_ptr<FermiMomentum> fermiMomentum;

public:
    CGlauberGenerator() = delete; // do not use default constructor
    CGlauberGenerator(const std::string&, const std::string&, double, bool, std::unique_ptr<FermiMomentum>&&);
    std::unique_ptr<cola::EventData> operator()() final;
};
```

```
class CGlauberFactory final: public cola::VFactory {
public:
    cola::VFilter* create(const std::map<std::string, std::string>&) final;

};
```

- Implementation details are concealed, interaction with the module happens only through overloaded call operator.
- Generator class is created by a separate factory class from XML readout.

https://github.com/apBUSampK/CGlauber



CMake integration

COLA/COLAConfig.cmake.in

```
@PACKAGE_INIT@

include(@CMAKE_INSTALL_PREFIX@/lib/cmake/COLA/COLAExport.cmake)
set_and_check(COLA_DIR @CMAKE_INSTALL_PREFIX@)
```

Module/CMakeLists.txt

```
find_package(COLA REQUIRED)
find_package(ROOT REQUIRED)

include(CMakePackageConfigHelpers)

set(CMAKE_INSTALL_PREFIX ${COLA_DIR})
```

Module/ModuleConfig.cmake.in

```
@PACKAGE_INIT@

find_package(COLA REQUIRED)
find_package(ROOT REQUIRED)

include(@CMAKE_INSTALL_PREFIX@/lib/cmake/CGlauber/CGlauberExport.cmake)
```

- Each library in the ecosystem is a CMake package.
- COLA package sets up the `COLA_DIR` variable for modules to be installed in the same directory.
- Module packages set up dependencies for the module.



Entire end-user program

CMakeLists.txt

```
# Find needed module packages
find_package(CGlauber)
find_package(CUniGen)
find_package(ExModule)
find_package(G4HandlerModule)
find_package(FermiBreakUp)

add_executable(COLATest main.cpp)

target_include_directories(COLATest PRIVATE ${COLA_DIR}/include)
target_link_libraries(COLATest PRIVATE CUniGen CGlauber ExModule G4HandlerModule FermiBreakUp)
```

main.cpp

```
int main() {
    // Create MetaProcessor and register needed filters (dependency injection)
    cola::MetaProcessor metaProcessor;
    metaProcessor.reg(std::unique_ptr<cola::VFactory>(new CGlauberFactory), "cglauber", cola::FilterType::generator);
    metaProcessor.reg(std::unique_ptr<cola::VFactory>(new CUniGenFactory), "cunigen", cola::FilterType::writer);
    metaProcessor.reg(std::unique_ptr<cola::VFactory>(new ExWFactory), "typer", cola::FilterType::writer);
    metaProcessor.reg(std::unique_ptr<cola::VFactory>(new G4HandlerFactory), "ablation", cola::FilterType::converter);

    // Assemble manager and run
    cola::ColaRunManager manager(metaProcessor.parse("../data/config.xml"));
    manager.run(1001);
    return 0;
}
```

config.xml

```
<?xml version="1.0" encoding="UTF-8" ?>
<program>
    <generator name="cglauber" name_a="O" name_b="O" energy="1e3" is_collider="1"/>
    <converter name="ablation" />
    <writer name="cunigen" fname="out.root" buff_size="100"/>
</program>
```

- Compact and simple.
- Dependencies are handled by CMake packaging.
- Main is set up only once for a set of required models.
- One can quickly switch between registered models by changing the config file.

Project backlog

- Library maintenance: stable DTO, documentation revamp. Release by the end of the year.
- New modules: UrQMD event file reader (BM@N requirement), SMASH, HepMC3, etc.
- Documentation for existing modules.
- Performance testing: comparison to existing monolith programs.
- Multi-threading support in manager class.

Summary

- Created prototype demonstrates possibilities of modularization in application to heavy-ion collision modelling.
- DI can be implemented in C++ with abstract factory pattern and allows for efficient dependency decoupling.
- CMake packages prove to be a useful tool for managing dependencies
- A foundation for COLA ecosystem has been set. While still lacking in modules, it can significantly simplify Monte Carlo modelling in future. We are open to collaborations and ready to help with model integration!

This work was supported by the Ministry of Science and Higher Education of the Russian Federation, Project FFWS-2024-0003.