Serialization and deserialization of complex data structures, and applications in high performance computing

by

Yaroslav Zaluzhnyi

from

Artemivsk, Ukraine

A Thesis submitted to the Faculty of Informatics, Chair of Systems Engineering of Technische Universität Dresden in partial fulfillment of the requirements for the degree of

Master of Science

Responsible Professor (1st reviewer)

Prof. Dr. Ivo Sbalzarini

2nd reviewer

Prof. Dr. Christof Fetzer

Dresden, 2016
Objectives (mandatory)

1. Understanding and reviewing (in the written report) the concepts behind the problem of serialization/de-serialization marshalling/unmarshalling of data structures;
2. Finding and analyzing available solutions in the literature and discussing them with pros and cons;
3. Discussing and listing available software implementations in other libraries;
4. Create and implement a serialization/deserialization implementation in OpenFPM that is compatible with the templated data structures and compile-time information in OpenFPM;
5. Implement in OpenFPM checkpoint restart (load and save) for parallel data-structures like Grid and Vectors using parallel IO in HDF5;
6. Create an interface that uses serialization and de-serialization to migrate data-structures across processors.

Optional

1. Implementation of a Particle-Mesh simulation as a test case to show an application at high level.
Declaration

I hereby certify that this Thesis has been composed by myself, and describes my own work, unless otherwise acknowledged in the text. All references and verbatim extracts have been quoted and all the sources of information have been specifically acknowledged. This Thesis has not been submitted or accepted in any previous application for a degree.

Furthermore, I declare that all rights in this Thesis are delegated to the Faculty of Informatics, Chair of Systems Engineering of Technische Universität Dresden.

________________________________________
Dresden, date

________________________________________
Signature
Abstract

The master thesis starts with the "serialization/deserialization" and "marshalling/unmarshalling" terms explanation and discussion of the differences between them. Problems of serialization and deserialization are reviewed and solutions to them are shown.

According to developed in the next chapter property lists, the suitable for this work software implementations of serializers are picked up, their functions and principles of work are discussed and compared with one another in terms of goals that are needed to achieve, and the most appropriate serializer is selected.

In the following chapter serialization/de-serialization missing functionality (which is a grid Packer/Unpacker) is developed and introduced in order for serializer to be completely implemented.

After that the checkpoint restart (load and save) for parallel data-structures (Grid and Vector) using parallel IO in HDF5 is designed, explaining the principles of work and algorithms.

As the last step an interface that uses serialization and de-serialization to migrate data-structures across processors (Grid map() function) is implemented and its step-by-step concept of work is shown.

All chapters are summarized in this report with appropriate code examples and necessary design. All the main objectives of the master thesis are reached.

A CD-ROM with all source code produced, input data used, output generated, source files of the report and all presentation slides are provided.
Table of Contents

1 Introduction........................................................................................................................................6
2 Serialization/deserialization and marshalling/unmarshalling of data ..............7
  2.1 Difference between serialization and marshalling..............................................7
  2.2 Problems and solutions of serialization/deserialization and marshalling/unmarshalling ........................................................................................................8
  2.3 Available existing software implementations..................................................9
    2.3.1 Boost::serialize ..............................................................................................10
    2.3.2 Avro ..............................................................................................................11
    2.3.3 Protobuf ......................................................................................................13
    2.3.4 Cap’n Proto ..................................................................................................15
    2.3.5 OpenFPM Packer/Unpacker .......................................................................19
3 Selection of the serializer .................................................................................................24
  3.1 Serialization-deserialization benchmark .........................................................24
  3.2 Integration of serializers with the templated data structures in OpenFPM......26
4 OpenFPM Packer/Unpacker for Grid ........................................................................29
5 OpenFPM checkpoint restart .......................................................................................31
  5.1 Checkpoint restart implementation .....................................................................31
  5.2 Checkpoint restart performance tests ..................................................................35
6 The map() function for OpenFPM Grid .................................................................40
7 Lists .................................................................................................................................43
  7.1 List of abbreviations ..............................................................................................43
8 References .........................................................................................................................44
1 Introduction

The Serialization of complex structures is the capability to convert data structures into a storing format or a simple sequence of bytes, and the possibility to be reconstructed later in the same or another computer environment (deserialization).

The natural context where serialization can be used in high performance computing is checkpoint restart. Checkpoint restart in high performance computing is the most trivial form of resilience. A program dumps its status at regular intervals and if the program terminates because of crashes of one or more nodes, it can be restarted from the previous saved state. The requirement of having data structures that can be serialized/deserialized, saved and reconstructed is a fundamental point to make checkpoint restart possible. Serialization can also be used in migration and exchange of data across processors. In a parallel distributed program running across several processors, each processor has only one part of the data to process. Each processor in order to process its part could require information stored on another processor. In other cases, processors that are computationally heavily loaded could want to migrate parts of their data to processor that are lightly loaded. In both cases serialization/deserialization of data structures is a fundamental prerequisite.

OpenFPM is a growing C++ template meta-programming library for particle mesh simulation, its core functionality is based on distributed data structures for development of numerical simulations on HPC systems. I am involved in creating a checkpoint restart interface for distributed data structures using serialization/de-serialization. Despite the capability of OpenFPM to exchange and migrate primitives or collections of primitives, the I will extend these capabilities. In particular, serialization/deserialization must be implemented in order to enable the already present parallel data-structures to store any kind of data, save the contained information and load it, migrate information for load rebalancing, and make retrieval/sending of information from/to other processors easier. Additionally, serialization/de-serialization components must simplify the construction of new parallel data structures with the previously implemented features.
2 Serialization/deserialization and marshalling/unmarshalling of data

2.1 Difference between serialization and marshalling

Marshaling and serialization are loosely synonymous in the context of remote procedure call, but semantically different as a matter of intent. Both do one common thing - that is serializing the object.

But as for serialization, when you serialize an object, only the member data within that object is written to the byte stream; not the code that actually implements the object. On the other hand, term "marshalling" is used when we talk about passing object to remote objects(RMI). In marshalling object is serialized (member data is serialized) and the codebase is attached. Codebase is the information that tells the receiver of object where the implementation of this object can be found. Any program that thinks it might ever pass an object to another program that may not have seen it before must set the codebase, so that the receiver can know where to download the code from, if it doesn't have the code available locally. The receiver will, upon deserializing the object, fetch the codebase from it and load the code from that location.

As the corresponding [1] Wikipedia page says: “The term "marshal" is considered to be synonymous with "serialize" in the Python standard library, but the terms are not synonymous in the Java-related RFC 2713:

To "marshal" an object means to record its state and codebase(s) in such a way that when the marshalled object is "unmarshalled", a copy of the original object is obtained, possibly by automatically loading the class definitions of the object. You can marshal any object that is serializable or remote. Marshalling is like serialization, except marshalling also records codebases. Marshalling is different from serialization in that marshalling treats remote objects specially.

To "serialize" an object means to convert its state into a byte stream in such a way that the byte stream can be converted back into a copy of the object."

So, since an OpenFPM library computations are done in parallel using HPC cluster of computers, the codebase is a priory known. Thus, in the content of this work, there is no difference in use of both – "serializing" or "marshalling" - terms.
2.2 Problems and solutions of serialization/deserialization and marshalling/unmarshalling

The complexity of serialization and marshaling of the data structure depends on a complexity of the data structure itself.

Serializing a primitive type like a “bool”, “int” or “float” is trivial: just write the data as it is (assuming that no compression is used). Serializing a pointer is more complex: the object it points to have to be serialized first. That way deserializing the pointer consists of setting its value to the memory address at which the object has been reconstructed. Moreover, the higher complexity of the pointer graph (graph is a set of trees, graph with/without loops), the higher difficulty of serialization to implement. As mentioned in [2] “A practical guide to C++ serialization”, there can be pointed out three levels of complexity in serialization, depending on how complex the pointer (and reference) graph is (see Figure 1 below):

1) The pointer graph is a forest (i.e., a set of trees);
2) The pointer graph is a directed acyclic graph (DAG), i.e., a graph without loop;
3) The pointer graph is a general graph, i.e., it may have loops.

Figure 1: {Pointer graph as a tree, a DAG and with loops}
First case is the simplest: when pointer graph is a tree, data can simply be serialized bottom up with a depth first traversal of the trees. In the case of directed acyclic graph, we can still serialize the data bottom up, making sure we write and restore shared data only once. When the pointer graph has loops, we need to write and restore data with forward references so that loops are handled properly. This is the hardest case to implement.

Analyzing OpenFPM library and in particular, the data types which are used for serialization/deserialization in the case of this master thesis, I see that there are no any pointers inside. Also, there is no question about which – binary or text – format to use: I see that in the fifth main goal of the thesis I have to provide a checkpoint restart using parallel IO in HDF5, and the HDF5 files are binary files. This means that I don't need to care about graph complexity and can serialize and deserialize the data as a stream of byte “as it is” into a memory.

2.3 Available existing software implementations

There are several common serialization/deserialization implementations in other external libraries.

I created the first “rough” list of requirements for selecting the most suitable of them in terms of this thesis:

1) A library should have serialization of STL containers and other commonly used templates.
2) Interface that is intuitively simple to use and understand.
3) Good documentation base.

Applying these requirements to more than 10 most common software implementations in the internet I picked up the most satisfying ones:

1. Boost::serialize;
2. Avro;
3. Protobuf;
4. Cap’n Proto;
5. OpenFPM Packer/Unpacker.
2.3.1 Boost::serialize

[3] Boost::serialize is a powerful and, probably, the most common library for C++ serialization. Has a good documentation base, intuitively understandable interface and wide range of functionalities.

As the Boost::serialize [4] description says, "a type T is Serializable if and only if one of the following is true:

- it is a primitive type.
  By primitive type we mean a C++ built-in type and ONLY a C++ built-in type. Arithmetic (including characters), bool, enum are primitive types.
- It is a class type and one of the following has been declared according to the prototypes detailed below:
  - a class member function „serialize“
  - a global function „serialize“
- it is a pointer to a Serializable type.
- it is a reference to a Serializable type.
- it is a native C++ Array of Serializable type."

The interface of Boost::serialize is pretty simple: for primitive types, the template operators & << > (see Figure 2 below) of the archive classes described above will generate code to save/load all primitive types to/from an archive. This code will usually just add the data to the archive according to the archive format. For example, a four byte integer is appended to a binary archive as 4 binary bytes while a to a text archive it would be rendered as a space followed by a string representation.

For class/struct types, the template operators & <<, and >> will generate code that invokes the programmer's serialization code for the particular data type. There is no default. An attempt to serialize a class/struct for which no serialization has been explicitly specified will result in a compile time error. The serialization of a class can be specified via either a class member function or a free function which takes a reference to an instance of the class as an argument.
When these operators are invoked for primitive data types, the data is simply saved/loaded to/from the archive. When invoked for class data types, the class „serialize“ function (see Figure 3 below) is invoked. Each „serialize“ function uses the above operators to save/load its data members. This process will continue in a recursive manner until all the data contained in the class is saved/loaded.

Boost::serialize also has non-intrusive version of serializer and supports derived classes.

**2.3.2 Avro**

[5] Avro relies on schemas. When Avro data is read, the schema used when writing it is always present. This permits each datum to be written with no per-value overheads, making serialization both fast and small. This also facilitates use with dynamic, scripting languages, since data, together with its schema, is fully self-describing. When Avro data is stored in a file, its schema is stored with it, so that files may be processed later by any program. If the program reading the data expects a different schema this can be resolved, since both schemas are present. Avro schemas are defined with [6] JSON file format (see Figure 4 below). This facilitates implementation in languages that already have JSON libraries.
Serialization and deserialization of complex data structures, and applications in high performance computing

Fakultät Informatik Distributed Systems Engineering

A JSON file is built on two structures:

- A collection of name/value pairs. In various languages, this is realized as an object, record, struct, dictionary, hash table, keyed list, or associative array.
- An ordered list of values. In most languages, this is realized as an array, vector, list, or sequence.

An object is an unordered set of name/value pairs. An object begins with { (left brace) and ends with } (right brace). Each name is followed by : (colon) and the name/value pairs are separated by , (comma). An array is an ordered collection of values. An array begins with [ (left bracket) and ends with ] (right bracket). Values are separated by , (comma). A value can be a string in double quotes, or a number, or true or false or null, or an object or an array. These structures can be nested. A string is a sequence of zero or more Unicode characters, wrapped in double quotes, using backslash escapes. A character is represented as a single character string. A string is very much like a C or Java string. A number is very much like a C or Java number, except that the octal and hexadecimal formats are not used.

```json
{
  "type": "record",
  "name": "Record",
  "fields": [
    {
      "name": "ids",
      "type": {
        "type": "array",
        "items": "long"
      }
    },
    {
      "name": "strings",
      "type": {
        "type": "array",
        "items": "string"
      }
    }
  ]
}
```

Figure 4: { JSON file }

A Schema is represented in JSON by one of:

- A JSON string, naming a defined type.
• A JSON object, of the form: {"type": "typeName" ...attributes...} where typeName is either a primitive or derived type name, as defined below. Attributes not defined in this document are permitted as metadata, but must not affect the format of serialized data.

• A JSON array, representing a union of embedded types.

Avro data is always serialized with its schema. Files that store Avro data should always also include the schema for that data in the same file. Because the schema used to write data is always available when the data is read, Avro data itself is not tagged with type information. The schema is required to parse data. In general, both serialization and deserialization proceed as a depth-first, left-to-right traversal of the schema, serializing primitive types as they are encountered. Avro specifies two serialization encodings: binary and JSON.

2.3.3 Protobuf

[7] Protocol buffers are Google's language-neutral, platform-neutral, extensible mechanism for binary serialization of structured data. User defines how he wants his data to be structured once, then he can use special generated source code to easily write and read structured data to and from a variety of data streams and using a variety of languages.

Protobuf, like Avro, uses special file format ("proto", see Figure 5 below). You specify how you want the information you’re serializing to be structured by defining protocol buffer message types in .proto files. Each protocol buffer message is a small logical record of information, containing a series of name-value pairs.
Once the messages are defined, you run the protocol buffer compiler for C++ on your .proto file to generate data access classes. The compiler generates a .h (the header which declares generated classes) and .cc (which contains the implementation of classes) files from each .proto, with a class for each message type described in your file. These provide simple accessors for each field as well as methods to serialize/parse the whole structure to/from raw bytes (Figure 6).

```
message Person {
  required string name = 1;
  required int32 id = 2;
  optional string email = 3;
}
enum PhoneType {
  MOBILE = 0;
  HOME = 1;
  WORK = 2;
}
message PhoneNumber {
  required string number = 1;
  optional PhoneType type = 2 [default = HOME];
}
repeated PhoneNumber phone = 4;
```

Figure 5: { Sample .proto structure }

```
Person person;
person.set_name("John Doe");
person.set_id(1234);
person.set_email("jdoe@example.com");
fstream output("myfile", ios::out | ios::binary);
person.SerializeToOStream(&output);
```

Figure 6: { Accessors and methods provided by generated classes }
Then, later on, the message could be read back in (Figure 7):

![Reading back a protobuf message](image)

The protocol buffer format supports the idea of extending the format over time in such a way that the code can still read data encoded with the old format.

### 2.3.4 Cap’n Proto

[8] Cap’n Proto is a data interchange format and capability-based RPC system. It is also a binary format. The distinctive feature of this system is that there is no encoding/decoding step. The Cap’n Proto encoding is appropriate both as a data interchange format and an in-memory representation, so once the structure is built, the bytes could simply be written straight out to disk.

Here the encoding is defined byte-for-byte independent of any platform. However, it is designed to be efficiently manipulated on common modern CPUs. Data is arranged like a compiler would arrange a struct – with fixed widths, fixed offsets, and proper alignment. Variable-sized elements are embedded as pointers. Pointers are offset-based rather than absolute so that messages are position-independent.

Similarly to Protocol Buffers, Cap’n Proto generates classes with accessor methods that you use to traverse the message. But in this case, these accessors validate pointers before following them. If a pointer is invalid (e.g. out-of-bounds), the library can throw an exception or simply replace the value with a default / empty object. Thus, Cap’n Proto checks the structural integrity of the message.

Like Protobufs, Cap’n Proto messages (Figure 8 below) are strongly-typed and not self-describing. The message structure should be defined in a special language, then invoked the Cap’n Proto compiler (“capnp compile”) to generate source code to manipulate that message type in desired language (C++ in my case). For example, running “capnp compile -oc++ myproto.capnp” will create `myproto.capnp.h` and `myproto.capnp.c++` in the same directory as `myproto.capnp`.

"Serialization and deserialization of complex data structures, and applications in high performance computing"
Let's take a more detailed look on how the serializing with a pointer to the message works. The unit of communication in Cap’n Proto is a “message”. A message is a tree of objects, with the root always being a struct. Physically, messages may be split into several “segments”, each of which is a flat blob of bytes. Typically, a segment must be loaded into a contiguous block of memory before it can be accessed, so that the relative pointers within the segment can be followed quickly. The first word of the first segment of the message is always a pointer pointing to the message’s root struct. Each segment in a message contains a series of objects. An “object” is any value which may have a pointer pointing to it. Pointers can only point to the beginning of objects, not into the middle, and no more than one pointer can point at each object. Thus, objects and the pointers connecting them form a tree, not a graph. An object is itself composed of primitive data values and pointers, in a layout that depends on the kind of object.

```
interface Node {
  isDirectory @0 () -> (result :bool);
}

interface Directory extends(Node) {
  list @0 () -> (list: List(entry));
  struct Entry {
    name @0 :Text;
    node @1 :Node;
  }
  create @1 (name :Text) -> (file :File);
  mkdir @2 (name :Text) -> (directory :Directory);
  open @3 (name :Text) -> (node :Node);
  delete @4 (name :Text);
  link @5 (name :Text, node :Node);
}

interface File extends(Node) {
  size @0 () -> (size: UInt64);
  read @1 (startAt :UInt64 = 0, amount :UInt64 = 0xffffffffffffffff) -> (data: Data);
  # Default params = read entire file.
  write @2 (startAt :UInt64, data :Data);
  truncate @3 (size :UInt64);
}
```

Figure 8: { .capnp file format }

- Types come after names.
- The @N annotations show how the protocol evolved over time, so that the system can make sure to maintain compatibility with older versions. Fields (and enumerants, and interface methods) must be numbered consecutively starting from zero in the order in which they were added. Unlike Protobufs, numbers cannot be skipped when defining fields.
The following types are automatically defined:

1) Void: Void;
2) Boolean: Bool;
3) Integers: Int8, Int16, Int32, Int64;
4) Unsigned integers: UInt8, UInt16, UInt32, UInt64;
5) Floating-point: Float32, Float64;
6) Blobs: Text, Data;
7) Lists: List(T).

Other existing complex types include:

- **Structure**, which has a set of named, typed fields, numbered consecutively starting from zero. Fields can have default values.
- **Unions**. It is two or more fields of a structure which are stored in the same location. Only one of these fields can be set at a time, and a separate tag is maintained to track which one is currently set.
- **Groups**, which are sets of fields that are encapsulated in their own scope (groups on their own are useless. They become interesting when used together with unions).
- **Dynamically-typed fields**. A struct may have a field with type “AnyPointer”. This field’s value can be of any pointer type – i.e. any struct, interface, list, or blob. (This is essentially like a “void*” in C).
- **Enums**, which are types with a small finite set of symbolic values.
- **Interfaces**. An interface has a collection of methods, each of which takes some parameters and return some results. Like struct fields, methods are numbered. Interfaces support inheritance, including multiple inheritance.
- **Annotations**. There is a possibility to attach an extra information to parts of the protocol that isn’t part of the Cap’n Proto language. This information might control details of a particular code generator, or might even be read at run time to assist in some kind of dynamic message processing.
- **Generic Types** (Figure 9). A struct or interface type may be parameterized, making it generic, similarly to C++ templates.
- **Generic Methods** (Figure 10). Interface methods may also have “implicit” generic parameters that apply to a particular method call.
Constants can also be defined in Cap’n Proto. These don’t affect what is sent on the wire, but they will be included in the generated code. Constants, aliases, and type definitions can be nested inside structs and interfaces. This has no effect on any definition involved except to define the scope of its name. Primitive constants are just constexpr values. Pointer-type constants (e.g. structs, lists, and blobs) are represented using a proxy object that can be converted to the relevant Reader type.

Cap’n Proto package also includes a command-line tool called “Capnp” intended to aid development and debugging. This tool can be used to:

- Compile Cap’n Proto schemas to produce source code in multiple languages.
- Generate unique type IDs.
- Decode Cap’n Proto messages to human-readable text.
- Encode text representations of Cap’n Proto messages to binary.
- Evaluate and extract constants defined in Cap’n Proto schemas.

Figure 9: { A generic structure with parameters }

Figure 10: { Here the method newAssignable() is generic. The return type of the method depends on the input type }
2.3.5 OpenFPM Packer/Unpacker

OpenFPM Packer/Unpacker is a part of OpenFPM library for particle mesh simulation, within the context of which this work in written. The idea of OpenFPM Packer/Unpacker is to “pack” data into a memory subsequently, with the possibility to “unpack” it from sequence of bytes into a pre-allocated object on the receiver side (Figure 11 below):

```cpp
// An object to pack
int num;

// Pack request
size_t req = 0;
Packer<decltype(num),HeapMemory>::packRequest(num,req);

// Allocate the memory
HeapMemory pmem;
ExtPreAlloc<HeapMemory> & mem = *(new ExtPreAlloc<HeapMemory>(req,pmem));
mem.incRef();

// Packing
Pack_stat sts;
Packer<decltype(num),HeapMemory>::pack(mem,num,sts);

// Unpacking
Unpack_stat ps;
decltype(num) num_unp;
Unpacker<decltype(num_unp),HeapMemory>::unpack(mem,num_unp,ps);
```

**Figure 11**: { An example of OpenFPM Packer/Unpacker usage }

Consider having a data and a desire to transfer it to another processor of the local computer or to another computer in the cluster. In this example (Figure 11) the data is an integer `num`.

The OpenFPM Packer/Unpacker algorithm steps are:

1) Request the “packing”;
2) Allocate memory;
3) “Pack” the data;
4) “Unpack” the data.

A “packRequest” function returns a size of an object to pack as a `size_t req` variable, which is declared and initialized to zero before, and passed. It represents how many
bytes in memory is needed to be allocated to store the data. Pack request requires a datatype of an object (decltype(num) here).

The type of the memory, where the allocation is done, can be specified (HeapMemory, CudaMemory etc.). After creating an ExtPreAlloc object of pre-allocated memory it is resized according to the requested size of an object to pack. Then the smart reference counter is incremented.

The function “Pack” is actually where the data is being packed into an allocated memory. It requires a packing information object (sts here).

The “Unpack” function follows the similar interface: the pre-allocated object of the same type as the data passed is declared, as well as the packing information (ps here), and given to the unpacker together with allocated memory. Unpacker uses the pointer that points to the appropriate object. The datatype on the unpacking side is automatically known, since the OpenFPM library uses computers in one computational cluster.

OpenFPM Packer/Unpacker is generic in terms of a datatype. It supports packing/unpacking of all the fundamental C++ types, containers, as well as templated “openfpm” types (i.e. openfpm::aggregate, openfpm::vector, openfpm::grid, openfpm::Box etc.), so it is compatible with compile-time information of this library. The selection of the appropriate packer depending on the datatype of the packed data is also done at compile time.

Datatype, which is passed to the Packer as a template parameter, uses header file Pack_selector.hpp (Figure 12). It is a template metaprogramming selector, which at the compile time calculates a value depending on the datatype.
Each value has corresponding macro defined.

Finally, these macros are used to choose suitable partial implementation of packer (Figure 13).

This is a common and effective practice for creating partial implementations of a class or a struct in C++. 

Figure 12: { A part of the Pack_selector.hpp, showing implementation of making packer selection at the compile time }
Thus, the appropriate packer implementation for the exact datatype is chosen at the compile time, and the data is packed at runtime. The same concept is used by unpacker respectively.

More on OpenFPM datatypes:

`openfpm::aggregate` is one of the most used types in OpenFPM. This is the templated data structure. The base for the aggregate is `boost::fusion::vector<list...>`, as well as for most of the datatypes used to be packed. Basically, they are templated analogs of a usual structure. The “properties” from the list of `boost::fusion::vector<list...>` could be specified, being passed as a template parameters.

OpenFPM Packer/Unpacker supports the feature of packing/unpacking of the templated data structures with specified properties. This can be done by specifying them in the “< >” brackets (the template parameters of the packer and unpacker calls). In order to successfully unpack this data, the properties should be the same in packer and unpacker (Figure 14 below).
Not specifying properties by leaving empty brackets “<>” or not putting brackets at all when packing/unpacking templated data will pack and unpack all the properties from the list.

OpenFPM Packer/Unpacker also supports the packing of a OpenFPM data of any complexity of nesting objects (for example, `openfpm::aggregate<openfpm::vector<openfpm::grid>>`, etc). On packing, firstly the size of outer object is packed. Then packer goes inside into the inner objects, packs them, if they are packable, or packs the size and goes inside again and so on. This way the data and the metadata about sizes is sequentially packed. Unpacked follows the same schema. Pointers in this case have offsets, which are gotten from packing information object.

After unpacking is done, the packed memory should be destroyed and reference counter decremented (`mem.decRef(); delete &mem`).
3 Selection of the serializer

In order to select the most suitable serializer for this work, the two more concrete and important properties are considered. The serializer should:

1) Have relatively fast performance.
2) Be the most suitable one to be integrated with the templated data structures and compile-time information in OpenFPM;

3.1 Serialization-deserialization benchmark

In accordance with [9] “Benchmark comparing various data serialization C++ libraries”, we have the following bar chart in terms of speed of serialization and deserialization of given software implementations (Figures 15 and 16).

![Figure 15: (Serialization-deserialization speed for various serializers)](image)
Serialization and deserialization of complex data structures, and applications in high performance computing

In this comparison following results were obtained running 1000000 serialize-deserialize operations 50 times and then averaging results.

For Cap’n Proto, since it already stores data in a “serialized” form and serialization basically means getting pointer to the internal storage, full build/serialize/deserialize cycle was measured. In the case of other libraries serialize/deserialize cycle of the already built data structure was measured.

Speed matters in high performance computing. Cap’n Proto shows fastest results, and this is not a surprise: as mentioned before in Chapter 2.3.4 “Cap’n Proto”, it uses raw pointers to get the data, so there is no encoding step.

Figure 16: { Serialization-deserialization speed for Cap’n Proto and Flatbuffers }

"Serialization and deserialization of complex data structures, and applications in high performance computing"
In order to compare this data with OpenFPM Packer/Unpacker, I created the performance test for it. It uses the same object as was used in “Benchmark comparing various data serialization C++ libraries” (std::vector<int64_t> with 100 values) and the same number of serialize-deserialize operations and tests.

The result is shown on the Figure 17.

As we see, OpenFPM Packer/Unpacker performs even faster than Cap’n Proto and much faster than then other serializers. This is understandable, since Packer/Unpacker also uses pointers to a byte sequence in memory.

Considering results, the two favorites in terms of performance time are picked: OpenFPM Packer/Unpacker and Cap’n Proto.

### 3.2 Integration of serializers with the templated data structures in OpenFPM

As said in the “Objective 4” of this master thesis, the serialization/deserialization implementation in OpenFPM should be compatible with the templated data structures and compile-time information in OpenFPM.

OpenFPM Packer/Unpacker is naturally coded to be used in the OpenFPM library. It supports all its datatypes. Moreover, in most cases data of exactly these types is used for computations.

As an opposite, Cap’n Proto or any other listed above software implementations do not support the OpenFPM datatypes.
Thus, another problem of using all the presented serializers excepting OpenFPM Packer/Unpacker is the requirement of having a parser for data for any of them. In order for considered serializers to be used in terms of this work, the data should be converted into their individual data format. The schema of serialization and deserialization in this case would be (Figure 18):

```
Serialize:

OpenFPM Data ---> Parser ---> Serialization

Deserialize:

Deserialization ---> Parser ---> OpenFPM Data
```

Figure 18: { Serialization schema using listed (except Packer/Unpacker) serializers }

While using OpenFPM Packer/Unpacker the schema is (Figure 19):

```
Serialize:

OpenFPM Data ---> Serialization

Deserialize:

Deserialization ---> OpenFPM Data
```

Figure 19: { Serialization schema using Packer/Unpacker }
Creating a data parser for special serializers’ data format in terms of this thesis would make sense only in the case of significant serialization/deserialization speed advantage of other serializer(s) over OpenFPM Packer/Unpacker. But the picture of performance speed is reversed. Thus, coding a parser for data in this case is impractical.

Another reason is, to create an interface which could support partial serialization/deserialization of data (templated OpenFPM structures with specified properties) is hard (or for some serializers even impossible) to code and is extremely impractical, considering that there is already a tool OpenFPM Packer/Unpacker which is as fast (or faster) as any of software instruments I considered above.

In this way, OpenFPM Packer/Unpacker is the most suitable of reviewed serializers and is picked to implement next goals of this work.
4 OpenFPM Packer/Unpacker for Grid

Up to the time of creating present master thesis, the OpenFPM Packer/Unpacker was implemented for all the OpenFPM data types excepting Grid. Implementing packer and unpacker for Grid is the part of “Objective 4”.

The functionality of the grid packer is analogic to one for other types, that was described in Chapter 2.3.5 “OpenFPM Packer/Unpacker”. The interface is also similar. The pack request, memory allocator, packer and unpacker for the grid object are called successively (Figure 20).

```c
// An object to pack
grid_cpu<3, aggregate<float, size_t, grid_cpu<3, Point_test<float>>>> grid;

// Pack request
size_t req = 0;
Packer<dec_type(grid), HeapMemory>::packRequest<1,2>(grid, req);

// allocate the memory
HeapMemory pmem;
ExtPreAlloc<HeapMemory> & mem = *(new ExtPreAlloc<HeapMemory>(req, pmem));
mem.incRef();

// Packing
Pack_stat sts;
Packer<dec_type(grid), HeapMemory>::pack<1,2>(mem, grid, sts);

// Unpacking
Unpack_stat ps;
grid_cpu<3, aggregate<float, size_t, grid_cpu<3, Point_test<float>>>> grid_unp;
Unpacker<dec_type(grid_unp), HeapMemory>::unpack<1,2>(mem, grid_unp, ps);
```

![Figure 20: Packer and unpacker for grid_cpu<>](image)

Here the grid_cpu with dimensionality of 3 and the contained objects of type aggregate is packed and unpacked. This aggregates have types float, size_t and another grid_cpu<3,Point_test<float>>>. Note that only last two properties of an object are passed.

The size of the data is calculated and passed to the packer. Member "pack" (Figure 21 below) calls, in its turn, either pack_simple_cond function if the object is simple (which packs the data it is), or packs the size of the outer grid and calls call_aggregatePack for each aggregate, depending on the complexity of the inner object.
If the inner object is simple and not nested, the size of outer grid is packed and data is copied into the memory, considering specified properties. If the inner object is complex and/or has other complex (those which have member “pack” in their classes) objects inside, the call_aggregatePack calls packer recursively, packing the sizes of outer grids / vectors / aggregates, until it reaches the simple object to pack.

The Unpacker follows the same concept. The interface is similar and easy to understand after learning one for Packer.

The tests for Grid packing and unpacking is provided on the attached CD-ROM.

After designing a grid packer, OpenFPM Packer/Unpacker is completely implemented for any needed OpenFPM data structure at the moment.
5 OpenFPM checkpoint restart

5.1 Checkpoint restart implementation

Once the OpenFPM Packer/Unpacker is completely implemented, the checkpoint restart (load and save) for parallel data-structures like Grid and Vectors can be considered. The format to save the data is [10] HDF5. HDF5 is a data model, library, and binary file format for storing and managing data. It is designed for flexible and efficient I/O and for high volume and complex data.

The data should be saved in parallel, using in the process all involved processors. The checkpoint restart concept schema is (Figure 22):

![Checkpoint restart schema](image)

**Figure 22**: {Checkpoint restart concept schema}

*Parallel Vector (distributed vector) and parallel Grid (distributed grid) are OpenFPM data types that can use one or more processors to be stored and processed on. The distributed vectors typical initialization looks like this (Figure 23),

```c
// Vector of particles
vector_dist<dim, St, prop> vd(k, box, bc, g);
```

**Figure 23**: {Initializing the distributed vector}

where:

- `dim` – dimensionality of the space;
- `St` – space type (float, int etc.);
- `prop` – properties the vector element store in OpenFPM data structure format;
- `k` – number of particles;
- `box` – domain where the vector of elements lives;
• \(bc\) – boundary conditions (periodic or non-periodic);
• \(g\) – ghost extension.

The data members of this class, that contain the needed data to save, are \(v\_pos\) and \(v\_prp\) vectors (Figure 24),

which store positions in space and properties of particles respectively. The metadata needed (in this case this is a number of processors involved in computations before saving) is also saved.

The “save” member of \(vector\_dist\) (distributed vector class) starts with a packing of needed vectors consequently into a memory. An MPI information in collected and the new HDF5 file with parallel processors’ read/write access is created (Figure 25).

\begin{verbatim}
int mpi_rank = v_cl.getProcessUnitID();
int mpi_size = v_cl.getProcessingUnits();
MPI_Comm comm = v_cl.getMPIComm();
MPI_Info info = MPI_INFO_NULL;

// Set up file access property list with parallel I/O access
hid_t plist_id = H5Pcreate(H5P_FILE_ACCESS);
H5Pset_fapl_mpio(plist_id, comm, info);

// Create a new file collectively and release property list identifier.
hid_t file = H5Fcreate (filename.c_str(), HSF_ACC_TRUNC, HSP_DEFAULT, plist_id);
H5Pclose(plist_id);
\end{verbatim}

Figure 25 : { MPI global information gathered and HDF5 file created }

After that, respecting an interface, goes HDF5 routine: “dataspaces” are created in the file and in the memory; appropriate “datasets” are created resizing them according to the
byte sequences’ sizes for data and metadata. The offsets for each processors write is calculated and, finally, the data is written. The resources now should be closed/released.

The “load” function has the same concept. The MPI global information is gathered and the HDF5 file opened for collective parallel read. The dataset for metadata is accessed and the metadata is read. The dataset for data is accessed and the data is read following the offset, calculated using the metadata. But what happens when the number of processors, loading HDF5 file is different from the number of processors saving it?

The algorithm is following: if the load is done on more processors, the data is loaded for the appropriate processors, and others remain empty. If the load is done on less processors, the “old” number of processors is divided by a new one. The data from the number of old processors, equivalent to the integer from a division, is assigned to each new processor. Then the data from the number of old processors, equivalent to the remainder of the division, is distributed across new processors one by one, starting from the one which has the smallest ID. For example: if the data is saved on 2 processors, loaded on 3 - two of new ones take the data, one remains empty. If the data is saved on 5 processors, loaded on 3 - each of new ones take \( \text{int}(5/3) = 1 \) – number of processors to load data from, and the remainder of the division \( \text{rem}(5/3) = 2 \) indicates number of processors, whose data is assigned one by one to new processors successively. Using this algorithm, the pretty uniform distribution of data across processors is achieved.

The memory on each processor is allocated, resized by the corresponding size of data, and the data is read into it. The pointers are passed to Unpacker, and the objects are constructed from memory into pre-allocated objects of the same type.

Now, to finish reconstructing vector_dist the particles should be reassigned from one processor to another. This is achieved by map() function, which was written in OpenFPM earlier and is not the part of the thesis.

The distributed grid typical initialization looks like this (Figure 26),

```cpp
// Distributed grid with id decomposition
grid_dist_id<dim, St, T> grid_dist(sz, domain, g);
```

Figure 26: {Initializing the distributed grid}
- $T$ – type of the objects grid is storing;
- $sz$ – vector of grid sizes on each dimension;
- $domain$ - box that contain the grid.

The `save` and `load` functions for `grid_dist` (distributed grid class) are implemented the same way as ones for `vector_dist`, using the same algorithms, Packer/Unpacker and HDF5 functionality. The data in this case is $loc\_grid$ vector, containing data for the local grids, $gdb\_ext$ vector, containing information about grid boxes and points of origin and $sub\_domains$ – the vector of local sub-domains of the space decomposition (Figure 27):

```cpp
template<unsigned int dim, typename St, typename T, typename Decom>
class grid_dist_id : public grid_dist_id_comm<dim,St,T,Decomp>
{
    // Domain
    Box<dim,St> domain;

    // Ghost expansion
    Ghost<dim,St> ghost;

    // Local grids
    mutable openfpm::vector<device_grid> loc_grid;

    // Space Decomposition
    Decomposition dec;

    // Extension of each grid: Domain and ghost + domain
    openfpm::vector<GBoxes<device_grid::dims>> gdb_ext;

    // Local sub domains
    openfpm::vector<SpaceBox<dim, St>> sub_domains;
```

**Figure 27**: { Main data of the distributed grid }

In order to finish the reconstruction of the grid, the parts of the local grids should be reassigned to new processors according to a new decomposition, if the object is loaded on different number of processors. Then local grids should be reconstructed. This is done by the `map()` function for grid (see Chapter 6 “The map() function for OpenFPM Grid”).
5.2 Checkpoint restart performance tests

In order to obtain performance picture of the checkpoint restart, it was tested for distributed vector and grid.

Vector case.

In the case of vector, it was tested in 3D implementation with the number of particles 1000000. The save on 5 processors, and the next load on 3, 5, 10 processors has been run and benchmarked. The results are provided below (Figures 28, 29, 30, 31).

![Figure 28: Distributed vector save on 5 processors](image1)

![Figure 29: Distributed vector load on 3 processors](image2)
Serialization and deserialization of complex data structures, and applications in high performance computing

Figure 30: { Distributed vector load on 5 processors }

Figure 31: { Distributed vector load on 10 processors }
Grid case.

In the case of grid, it was tested in 2D implementation with the size of 1000x1000 points. The save on 5 processors, and the next load on 3, 5, 10 processors has been run and benchmarked. The results are provided below (Figures 32, 33, 34, 35).

Figure 32 : { Distributed grid save on 5 processors }

Figure 33 : { Distributed grid load on 3 processors }
Serialization and deserialization of complex data structures, and applications in high performance computing

Figure 34: Distributed grid load on 5 processors

Figure 35: Distributed grid load on 10 processors
The output objects for vector and grid have sizes of 24 and 4 Mb respectively. The output files, as shown by operating system, have 22.9 and 3.8 Mb sizes respectively. Apparently, the reason of that is an HDF5 encoding and compression of the data and the way of counting the file size on the hard drive by Ubuntu 14.04.

In the both cases (vector and grid), when loading on 3 processors (in general, loading on the smaller number of processors than saving), two (in general, some) of the processors show slower results than others. That happens because of combining the data from several “old” processors into some of “new” ones.

Overall, the speed and the file sizes of the implemented checkpoint restart is satisfactory for high performance computing.
6 The map() function for OpenFPM Grid

Once the grid is initialized, the three “Initialize” functions are called in the constructor (Figure 36).

\[
\text{grid_dist_id(const size_t \& gSz)[dim], c:domain(domain), ghost(g), dec(create_vclus)}
\]

\[
\text{InitializeCellDecomposer(gSz,p.bc);} \\
\text{InitializeDecomposition(gSz, p.bc);} \\
\text{InitializeStructures(gSz);}
\]

Figure 36: { Grid constructor }

- \text{InitializeCellDecomposer} – sets parameters to the cell decomposition of the grid;
- \text{InitializeDecomposition} – initializes the grid, decomposes global domain into local sub-domains;
- \text{InitializeStructures} – constructs grid boxes and local grids, using the space decomposition.

Thus, right after the allocating a new grid object to load the old grid into, we have a new space decomposition according to a new (if so) different number of processors (Figure 37), and corresponding grids and boxes for grids.

Figure 37: { The example of old and new space decomposition (from 4 to 6 processors) }
On this example (Figure 38) the intersection between old sub-domain of processor 1 (the whole square box) and new sub-domains of processors 1 and 2. The intersection grid corresponding to the shaded region should be moved from processor 1 to processor 2.

![Figure 38: The intersection between old and new sub-domains](image)

After loading a grid, the `map()` function (Figure 39) moves all the grids that does not belong to the local processor to the respective processor.

```c
void map(Box<dim,St> domain, Decomposition & dec, CellDecomposer_sm<dim,St,shift<dim,St>> & cd_sm, openfm::vector<device_grid> & m_grid)
{
    // Processor communication size
    openfm::vector<device_grid> & m_grid_recv;

    // Constructs a vector of grids to move
    labelIntersectionGridsProcessor(domain, dec, cd_sm, sub_domains_old, loc_grid, gdb_ext, m_grid, psc_sz);

    // Vector of number of sending grids for each involved processor
    openfm::vector<device_grid> & m_grid_recv;

    // Vector of ranks of involved processors
    openfm::vector<device_map> & m_grid;

    for (size_t i = 0; i < v_c1.getProcessingUnits(); i++)
    {
        if (m_grid.get(i).size() != 0)
        {
            proc_r.add1();
            proc_r.add(m_grid.get(i).size());
        }
    }

    // Send and receive grid parts
    v_c1.SendRecv(m_grid_recv, m_grid, proc_r, recv_map, recv_sz_map);
}
```

![Figure 39: The grid map() function](image)
Step-by-step the concept is:

1) Constructing a vector of grids to move in the function `labelIntersectionGridsProcessors`. The algorithm here is for each of the old sub-domains check if they intersect with new sub-domains. If yes, then for the appropriate intersection box the corresponding part of the old grid is found, iterated across and information copied into an allocated intersection grid. Then this grid is put into a `m_oGrid` vector to the position, equal to number of processor to send a grid to;

2) Preparing processor communication objects, allocating `m_oGrid_recv` to receive the data;

3) Calling `SSendRecv` member, which sends and receives multiple messages, in our case containing grids data;

4) Reconstructing the grid by creating a new one and copying old and new data in it, and replacing the old one.

Once the `map()` function is executed after loading, the grid is reconstructed on a new number of processors according to a new space decomposition.
7 Lists

7.1 List of abbreviations

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU</td>
<td>Central Processing Unit</td>
</tr>
<tr>
<td>DAG</td>
<td>Directed Acyclic Graph</td>
</tr>
<tr>
<td>HDF</td>
<td>Hierarchical Data Format</td>
</tr>
<tr>
<td>HPC</td>
<td>High Performance Computing</td>
</tr>
<tr>
<td>IO</td>
<td>In-Out</td>
</tr>
<tr>
<td>RMI</td>
<td>Remote Method Invocation</td>
</tr>
<tr>
<td>STL</td>
<td>Standard Template Library</td>
</tr>
</tbody>
</table>
8 References