POLITECNICO DI TORINO
Corso di Laurea in Computer Engineering

Tesi di Laurea Magistrale

Data access layer optimization of the Gaia data processing in Barcelona for spatially arranged data

Relatori:
Jaime Delgado
Claudio Passerone
Marcial Clotet Altarriba

Candidata:
Francesca Ronchini

APRILE 2017

There is no greater joy than to have an endlessly changing horizon, for each day to have a new and different sun.

A mia madre

Politecnico di Torino

Abstract

Department of Control and Computer Engineering

Data access layer optimization of the Gaia data processing in Barcelona for spatially arranged data

by Francesca Ronchini

Gaia is an ambitious astrometric space mission adopted within the scientific programme of the European Space Agency (ESA) in October 2000. It measures with very high accuracy the positions and velocities of a large number of stars and astronomical objects. At the end of the mission, a detailed three-dimensional map of more than one billion stars will be obtained. The spacecraft is currently orbiting around the L2 Lagrangian Point, 1.5 million kilometers from the Earth. It is providing a complete survey down to the 20^{th} magnitude. The two telescopes of Gaia will observe each object 85 times on average during the 5 years of the mission, recording each time its brightness, color and, most important, its position. This leads to an enormous quantity of complex, extremely precise data, representing the multiple observations of a billion different objects by an instrument that is spinning and precessing. The Gaia data challenge, processing raw satellite telemetry to produce valuable science products, is a huge task in terms of expertise, effort and computing power. To handle the reduction of the data, an iterative process between several systems has been designed, each solving different aspects of the mission.

The Data Analysis and Processing Consortium (DPAC), a large team of scientists and software developers, is in charge of processing the Gaia data with the aim of producing the Gaia Catalogue. It is organized in Coordination Units (CUs), responsible of science and software development and validation, and Data Processing Centers (DPCs), which actually operate and execute the software systems developed by the CUs. This project has been developed within the frame of the Core Processing Unit (CU3) and the Data Processing Center of Barcelona (DPCB).

One of the most important DPAC systems is the Intermediate Data Updating (IDU), executed at the Marenostrum supercomputer hosted by the Barcelona Supercomputing Center (BSC), which is the core of the DPCB hardware framework. It must reprocess, once every few months, all raw data accumulated up to that moment, giving a

higher coherence to the scientific results and correcting any possible errors or wrong approximations from previous iterations. It has two main objectives: to refine the image parameters from the astrometric images acquired by the instrument, and to refine the Cross Match (XM) for all the detections. In particular, the XM will handle an enormous number of detections at the end of the mission, so it will obviously not be possible to handle them in a single process. Moreover, one should also consider some limitations and constraints imposed by the features of the execution environment (the Marenostrum supercomputer). Therefore, it is necessary to optimize the Data Access Layer (DAL) in order to efficiently store the huge amount of data coming from the spacecraft, and to access it in a smart manner. This is the main scope of this project. We have developed and implemented an efficient and flexible file format based on Hierarchical Data Format version 5 (HDF5), arranging the detections by a spatial index such as Hierarchical Equal Area isoLatitude Pixelization (HEALPix) to tessellate the sphere. In this way it is possible to distribute and process the detections separately and in parallel, according to their distribution on the sky. Moreover, the HEALPix library and the framework implemented here allows to consider the data at different resolution levels according to the desired precision. In this project we consider up to level 12, that is, 201 million pixels in the sphere.

Two different alternatives have been designed and developed, namely, a *Flat solution* and a *Hierarchical solution*. It refers to the distribution of the data through the file. In the first case, all the dataset is contained inside a single group. On the other hand, the hierarchical solution stores the groups of data in a hierarchical way according to the HEALPix hierarchy.

The Gaia DPAC software is implemented in *Java*, where the HDF5 Application Programming Interface (API) support is quite limited. Thus, it has also been necessary to use the Java Native Interface (JNI) to adapt the software developed in this project (in *C* language), which follows the HDF5 *C* API. On the Java side, two main classes have been implemented to read and write the data: *FileHdf5Archiver* and *FileArchiveHdf5FileReader*. The Java part of this project has been integrated into an existing operational software library, *DpcbTools*, in coordination with the Barcelona IDU/DPCB team. This has allowed to integrate the work done in this project into the existing DAL architecture in the most efficient way.

Prior to the testing of the operational code, we have first evaluated the time required by the creation of the whole empty structure of the file. It has been done with a simple program written in *C* which, depending on the HEALPix level requested, creates the skeleton of the file. It has been implemented for both alternatives previously mentioned. Up to HEALPix level 6 it is not possible to notice a relevant difference. For level 7

onwards the difference becomes more and more important, especially starting with level 9 where the creation time is uncontrollable for the Flat solution. Anyhow, the creation of the whole file is not convenient in the real case. Therefore, in order to evaluate the most suitable alternative, we have simply considered the Input/Output performance.

Finally, we have run the performance tests in order to evaluate how the two solutions perform when actually dealing with data contents. Also the *TAR* and *ZIP* solutions have been tested in order to compare and appraise the speedup and the efficiency of our new two alternatives. The analysis of the results has been based on the time to write and read data, the compression ratio and the read/write rate. Moreover, the different alternatives have been evaluated on two systems with different sets of data as input. The speedup and the compression ratio improvement compared to the previously adopted solutions is considerable for both HDF5-based alternatives, whereas the difference between the two alternatives. The integration of one of these two solutions will allow the Gaia IDU software to handle the data in a more efficient manner, increasing the final I/O performance remarkably.

This thesis is organized in seven chapters. Chapter 1 is a short introduction to the mission, where motivations and objectives are defined, followed by a brief overview of the state of the art. Chapter 2 provides a more detailed explanation of the Gaia mission, where aims and goals are described in a more specific way. Details regarding the previous file format are also given in this chapter. Chapter 3 and 4 focus on the two libraries mainly used in this project: HEALPix library and HDF5 library. In particular, chapter 3 describes in a deeper way the tesselation of the sky using the HEALPix index, explaining how it is possible to take advantage of it in this project. Chapter 4 gives more information about the HDF5 file format and its applicability to Gaia/DPAC. Chapter 5 describes the software developed in this thesis: in particular, the two alternatives solutions are illustrated, focusing on the two main I/O operations, writing and reading. Furthermore, in this chapter it is also described the integration with the DpcbTool library by way of the JNI interface. Tests and Results are described and reported in the Chapter 6, where the results of the tests done on the two new developed alternatives are compared between them. Comparison between the new solutions and the previous ones are also made, in order to consider the most efficient file format to apply. Conclusions and future work are reported in Chapter 7.

Acknowledgements

The present work is the result of a collaboration between the Departament dAstronomia i Meteorologia, Institut de Cincies del Cosmos (ICCUB), Universitat de Barcelona (UB), Gaia Data Analysis and Processing Consortium (DPAC), the Barcelona Supercomputing Center (BSC) teams, and the Facultat d'Informtica de Barcelona (FIB), Universitat Politcnica de Catalunya - BarcelonaTech (UPC).

First of all, I would like to acknowledge my supervisor M. Clotet. It was a pleasure to work with you throughout this project. I have learned a lot working by your side and from your experience. You always provided positive encouragement, also when things were not going correctly. You were always present and willing to help me, especially in the more difficult times. All of your advice and your continuous positive support have been fundamental for the completion of this thesis, which is also yours. I could not have imagined a better guide for this thesis. Muchas gracias.

A big thank you also to my advisors J. Delgado, C. Passerone and J. Portell. Your recommendation and support have been essential to me. You all were always willing to help me and I really appreciated it. It was a pleasure to work with all of you.

I also want to thank the DPAC group, especially all the people working on the DPCB team. It has been an honor and a privilege to work in this amazing environment. Thank you also to the Barcelona Supercomputing Center for the support given to this project.

Thank you to all the new friends that I have met in this amazing experience in Barcelona. A special thank you to Francesca and Anica, with whom I shared everything here. You have been essential for me in this adventure. Alix, Jutsyna and Maite, I will never be able to thank you in the proper way. Your help, especially in the last weeks, has been fundamental for me. Love you guys, see you soon.

Il mio più grande ringraziamento va a mia madre Catia, la quale ha sempre creduto in me ed in ogni mio progetto, senza esitare mai ad aiutarmi e standomi vicina in ogni singolo istante, guidandomi sempre nella giusta direzione. E soprattutto grazie a te se tutto questo è stato possibile.

Non posso non ringraziare tutte le persone della mia famiglia, le quali, anche se non sempre approvano e condividono le mie scelte, sono sempre pronte a sostenermi ed aiutarmi in ogni momento, specialmente nei più difficili. Un grazie speciale a mio fratello Alessio, mia Zia Silvia, mia cugina Giorgia e Claudio.

Grazie a Sandra, mia seconda mamma ed amica, la quale mi ha vista crescere e accompagnata in ogni passo della mia vita, soprattutto i più importanti. Il tuo sostengo e la tua positività sono stati fondamentali, soprattuto durante questo percorso.

Vorrei ringraziare anche tutte le amicizie nate durante gli ultimi cinque anni a Torino, le quali sono state al mio fianco lungo la strada verso questo traguardo. Un grazie particolare a Francesco, conosciuto per puro caso e diventato una delle persone più importanti della mia vita.

Mai troverò le parole giuste per ringraziare Chiara, amica di una vita. Grazie per i consigli, per le mille telefonate, per le infinite chiacchierate, per tutto il tempo che mi dedichi, per essere sempre presente, per aver sempre creduto in me, per la sincerità, per le risate, e per tutti i pochi (purtroppo) ma speciali momenti che condividiamo. Sarò sempre al tuo fianco e mai smetterò di ringraziarti per essere sempre al mio.

Grazie a tutti voi!

Francesca

Contents

List of Figures

Chapter 1

Introduction

1.1 Gaia and DPAC

The Gaia mission is one of the most challenging European Space Agency (ESA) space missions. It was adopted within the scientific programme of the ESA in October 2000 and the satellite was launched in December 2013. The main goal of the mission is to obtain the most detailed three-dimensional map of our galaxy, with more than 1 billion stars. The aim is also to measure in a very accurate way the position, velocity, brightness and temperature of the stars and galactic objects observed. This will be done by observing on average 85 times each region of the sky during the 5 years of nominal mission duration.

The project is based on the same principles of the High Precision Parallax Collecting Satellite (HIPPARCOS) mission. The spacecraft orbits around the Second Lagrange Point (L2) in the Sun–Earth system. Thanks to its high performance instruments and the spin of the satellite around its own axis, which itself precesses at a fixed angle of 45 degrees with respect to the Sun, it is possible to have the complete view of the Sky. Fig. 1.1 shows a Mollweide projection of the sky in terms of average number of observations of Gaia. In this figure the scanning law of Gaia combined with a simulated model of the Galaxy can be clearly appreciated. It can be seen how the telescopes will scan the Galatic center more times than other regions.

The scope of the mission will lead to the acquisition of an enormous quantity of data, which is complex to handle and to interpret. In particular, one of the main challenges is

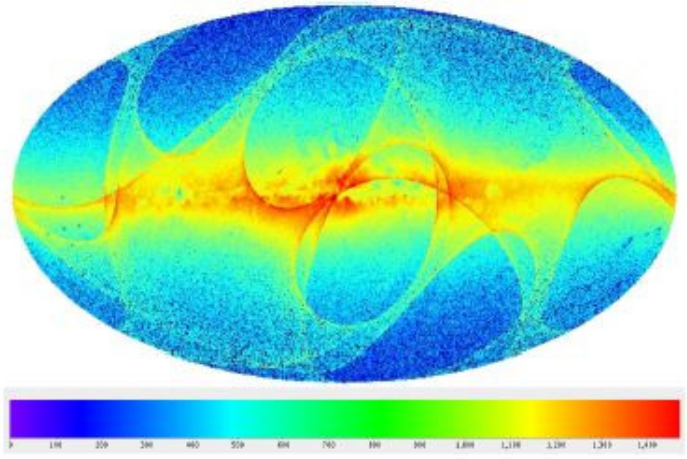

FIGURE 1.1: Mollweide projection of the sky showing the expected average number of observations per square degree.

to process the raw data into science products, which is the main task of the Data Analysis and Processing Consortium (DPAC). In fact, its main objective is the production of the final Gaia Catalogue. Behind this, there is a huge software project which uses Java as the baseline programming language. In order to store the data coming from the satellite, DPAC uses a custom file format where data is stored as serialized Java objects compressed in ZIP. One of the most important systems in DPAC is the Intermediate Data Updating (IDU), which has two main goals: to refine the image parameters for the astrometric information, and to refine the Cross Match (XM) for all the detections. This global XM is responsible of providing the links between the observations and the entries in the Gaia catalogue. The DPAC data processing and specifically the IDU XM are described in more detail in Chapter 2.

1.2 Motivation

The IDU XM is a process that requires to treat, in a single run, all of the accumulated observations. This will be a huge number at the end of the mission (around 10^{11} detections), so it will not be possible to process them sequentially. Instead, they will have to be distributed, following a very specific scheme, according to their spatial distribution. Afterwards, each process will be able to treat each sky region independently. To

perform this spatial distribution, the Hierarchical Equal Area isoLatitude Pixelization (HEALPix) library is used, which is described in detail in Chapter 3.

Another key factor will be the file format used to store the data, which should be more efficient than the default DPAC format. The choice has been to use the Hierarchical Data Format version 5 (HDF5) framework. In fact, this is a very powerful file format that allows to save and manage data very efficiently. More information on HDF5 is available in Chapter 4.

In any case, it is necessary to take into consideration some additional challenges, specially because in IDU it is not possible to use a database system. This is due to the architecture of the MareNostrum III supercomputer at the Barcelona Supercomputing Center (BSC), which has no central database. Storing the information in files divided by HEALPix index could lead to a huge number of files and therefore to very poor efficiency. On the other hand, due to the nature of the default DPAC file format, it would also be very inefficient to have very few files with a lot of information in each of them. Furthermore, we would also have poor performances, in the shared file-system environment of BSC MareNostrum, if we have several jobs performing I/O operations (specially write operations) on the same files. Finally, it is not easy to have files with quite uniform sizes because the distribution of the information depends on the distribution on the sky, which is strongly non-uniform as shown in Fig. 1.1.

There are other additional reasons that have motivated this thesis. The data retrieved from the satellite is stored as ZIP-compressed serialized data objects. The processing software and interfaces are written, by DPAC policy, in Java. Although the Java HDF5 API is currently limited, it is possible to use the Java Native Interface (JNI) to work with the *C* HDF5 API. It is clear that the JNI interface should be as simple as possible and the Data Access Layer (DAL) as efficient as possible. Fig. 1.2 shows an overview of the project scope.

1.3 Objectives

The aim of project is to define and implement the necessary Java JNI routines to properly use the *C* HDF5 code implementation in order to arrange and access the HEALPix data in an efficient manner.

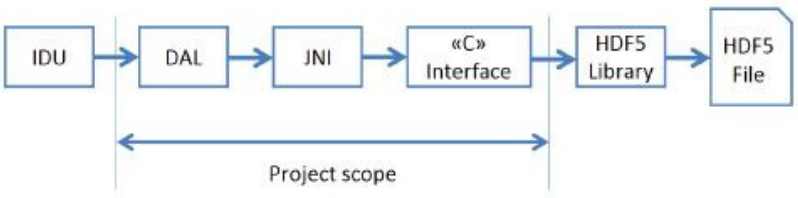

FIGURE 1.2: Project scope

In particular, an efficient data storage structure for spatially arranged data within the HDF5 file has to be defined first. To do so, the features of the data to be stored for the IDU XM must be considered. More than one solution will be implemented in order to evaluate the performance. One of them will use the HEALPix index name in order to store the data, leading to a hierarchical structure within the HDF5 file. The second one will use the HEALPix unique identifier to store the data, leading to a flat structure. Basically, the aim of the project is to develop and evaluate the best way to store the data in files, in order to retrieve them in the most efficient way. All the constrains and limitations described in Sect. 1.2 will be taken into account.

When the implementation of the C code and the JNI interface with the HDF5 are created, an interface with the current IDU DAL will be created. This part of the project will be done in coordination with the Barcelona IDU/Data Processing Center of Barcelona (DPCB) team, in order to facilitate the integration of the interfaces with the rest of the system. In particular, the aim is to implement the necessary Java code to read and write the information in the proper way, knowing how the data will be saved on the file from the C side. Thus, all the required native methods will be implemented in order to interact with the C API in the most efficient way.

As the final stage of the project, the solutions implemented will be analyzed in terms of I/O performance, since the scope of the project is the optimization of the data access. This will be done running performance tests with different scenarios in MareNostrum. Once performance figures are available, it will be possible to move on to the optimization phase to improve them if necessary.

All the software will be adequately documented, both as inline comments and as LaTeX-based documentation. The DPAC coding guidelines will be followed as coding style, good practices, and the use of unit tests as far as possible. Also, the code and documentation

will be stored in the central DPAC SubVersion repository for better coordination and integration with the rest of the team.

1.4 State of the art

HDF5 is not the only available solution, although it was considered the best option for this project. Other similar libraries and formats, well suited to store large quantities of data such as *N*-Dimensional Data Format (NDF), Advanced Scientific Data Format (ASDF) or Flexible Image Transport System (FITS) could be used. In this section, a brief overview of these is given. The idea is to provide a general idea about the pros and cons of each one. In any case, it is possible to find more information of NDF in [1], ASDF in [2] and FITS in [3] and [4].

The oldest of the three is the NDF, designed and developed in the 1980s, with its last release done in 2005. Its scope was to provide a data model specially for astronomy data processing applications. It also allows to organize and store arrays of structures and/or structures in a hierarchical way, which is self-described and can be queried. It is possible to merge structures in order to create arbitrary data types, and it is possible to modify the structures in the file if needed (modifying, deleting or copying them) [1]. There were some problems with the framework because it was not easy to convince the end users that hierarchical data structures were useful at all. Moreover, the library is written in Fortran, and it is not so easy to deal with it when having to interact with high-level interfaces (such as those written in Java, as in the case of this thesis). Furthermore, it does not support 64-bit dimensions or sizes, neither compression [1].

Another possible file format could be FITS. The first standardization was in the 1981, but the last release was in 2008. It was created specially as an image format, but the usage has been extended beyond this [4]. In fact, it is used for the transport, analysis and archival storage of scientific data sets. The possibility of data stored as human-readable ASCII with headers was one of the main features of the format, making the end users able to read information regardless of the provenance of the file [3]. This also allows the direct access to the data. In fact, it is also possible to save in the header offsets to some particular or important information in subsequent data units. The information in the header is stored using particular keywords (it is possible to think of them as

attributes). There are some constraints that should be considered. In particular, the size of the attribute names is limited, and there are restrictions regarding the size or type of values of the attributes. Moreover, the relations between the objects require special conventions. In spite of the limitations, the format is easily accessible in all major programming languages, and therefore it is not difficult to integrate it in projects.

ASDF could also considered as an alternative file format. In fact, we can consider it as an optimization of FITS. It is based on an existing text format that seems to remove the previous downsides of FITS. Contrary to FITS, it is possible to express the relations between the data, specially the complex ones, in an easier way. This is because the data and the metadata are cleverly interlaced. The data is intrinsically hierarchically structured, and it is possible to share references between different elements. Also, the data values are more flexible, and it is possible to have a more descriptive keyword name. However, the format is not able to handle very large amounts of distributed data in a efficient way, and it does not support chunking (for compression) [2].

Taking into account all these considerations, the HDF5 has been considered the best option for the implementation of this project.

Chapter 2

The Gaia mission

Gaia is the next astrometric mission of the European Space Agency (ESA), successor to the Hipparcos mission. It is built on the same conceptual principles but with cutting edge technologies. Fig. 2.1 shows an artistic impression of the satellite.

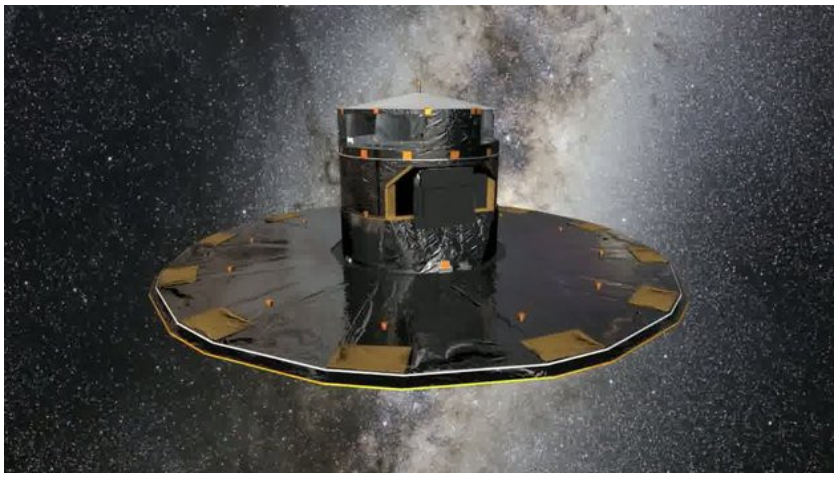

FIGURE 2.1: The Gaia spacecraft. (Credit: ESA)

The main goal of Gaia is to make the most precise three-dimensional map of our Galaxy, the Milky Way. However, this is not the only goal of the mission [5]. In fact, the spacecraft will allow to chart the position, velocity, distance and changes in brightness of more than 1.000 million stars (1% of the Milky Way). Moreover, it will be possible to determine temperature and chemical composition of many of them. The mission will also deliver information and very precise data regarding thousands of extra-solar planetary systems, huge number of smaller bodies never seen so far, distant quasars, thousands of supernovae and brown dwarfs [6].

The satellite was lunched on December 19th, 2013, from Kourou, in French Guiana. The nominal mission duration is five years. During this time, the satellite will observe each source about 85 times on average, recording each time information about color, brightness and especially position [7]. It means that the amount of data that is going to be collected is huge (over a Petabyte considering ground processing outputs) [8]. The data is sent compressed and coded from the spacecraft, therefore it is necessary to perform a complex data processing to make it scientifically meaningful.

Gaia operates at the L2 point of the Solar System [6]. This point is at about 1.5 million kilometers from the Earth in the anti-Sun direction, co-rotating around the Sun together with the Earth in its yearly orbit. Since this point is almost eclipse-free, it allows Gaia to observe the sky all year long without thermal disturbances. The spacecraft reached the operational orbit after three weeks from launch. After some months of commissioning, the satellite entered nominal operations in July 2014 [8]. The catalogue of the mission will be published in several releases, starting in autumn 2016.

2.1 The spacecraft

The spacecraft has been built by Airbus Defence and Space (ADS). Basically, it is composed of three main modules, as shown in Fig. 2.2.

- *Payload Module:* it contains the optical bench with the two telescopes and the science instruments (the astrometric instrument, the photometric instrument and the Radial-Velocity Spectrometer). It is protected by the thermal tent that ensures thermal stability, which is extremely important for the accuracy of the Gaia measurements.
- *Electrical Service Module:* it contains all the electronic modules required by the instruments (Payload Data Handling Unit, power module, communications units and other electronic subsystems).
- *Mechanical Service Module:* it contains all the mechanical, thermal and electronic elements to support all the instruments.

Gaia has the most precise Focal Plane ever launched before, made of 106 CCDs, reaching almost a billion pixels. The quality of the CCDs allows the instrument to work with very

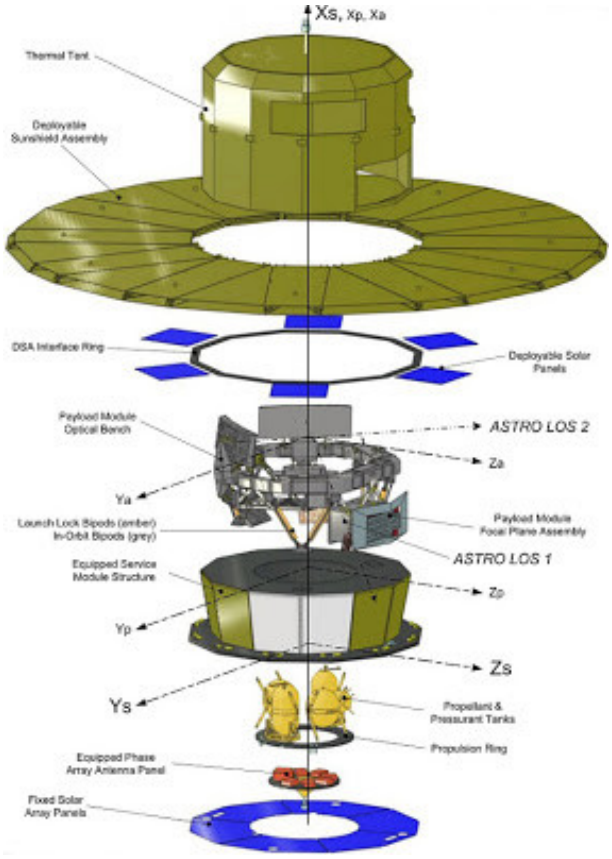

FIGURE 2.2: Main hardware modules of the Gaia spacecraft. (Credit: ESA/AirbusDS)

faint astrophysical magnitudes (up to the 20^{th} magnitude). The Focal Plane includes three instruments:

- *Astrometric Instrument,* used to measure the stellar positions, object flux and track their motion. After the adequate on-ground processing it will allow obtaining the star parallaxes.
- *Photometric Instrument,* whose task is to provide color information based on two low-resolution spectra (one on the blue band and one on the red band). It provides information regarding temperature, mass and chemical composition of the observed objects.
- *Radial Velocity Spectrometer,* used to determine the radial velocity and spectral signature of bright objects.

Fig. 2.3 shows a breakdown of the main hardware components of the spacecraft.

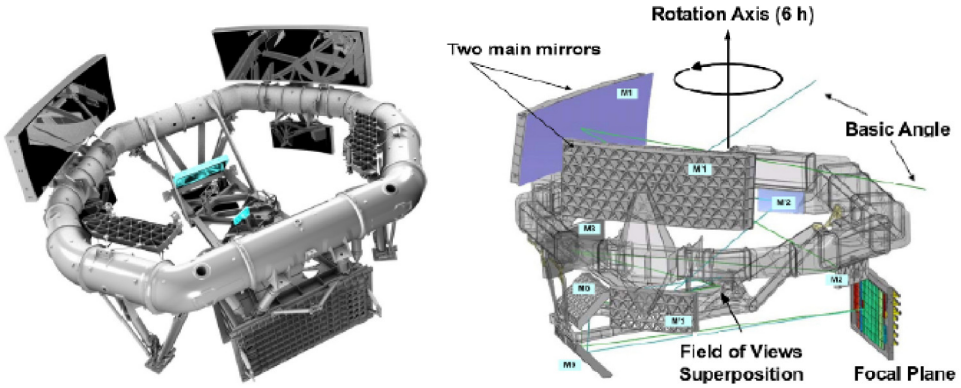

FIGURE 2.3: Overview of the hardware components of Gaia. (Credit: ESA/AirbusDS)

Gaia scans the sky in a continuous way, capturing images from the objects, combining the images from both telescopes onto the focal plane. The acquisition of the images is made using the Time Delayed Integration (TDI) mode. TDI is commonly used for imaging moving objects, where the camera is synchronized with the apparent speed of the object. Stellar objects are automatically detected by Gaia and their image collected, coded and afterwards downloaded to the ground segment.

2.2 Gaia Data Processing

Data as downloaded from the satellite cannot be directly used by scientists, so an adequate data reduction process has to be performed. For Gaia it consists of several phases which transform the data from the raw astrometric, photometric and spectroscopic telemetry to consistent and significant measures to be included in the catalogue.

The Gaia data reduction is a cyclic process organized in Data Reduction Cycle (DRC). In each DRC the previous results are used together with the new data to obtain a more refined product. DPAC is the European consortium in charge of managing the six different DPCs that process the Gaia data. Fig. 2.4 shows the cyclic data reduction process from the raw spacecraft telemetry until the delivery of the final catalogue.

Another important decision for DPAC was which programming language had to be used. The final decision was to use Java. The main reason that brought to this choice is the portability of the code. As the Gaia data processing is distributed into several DPCs, which have different architectures, portability was a concern. In fact, it is not possible

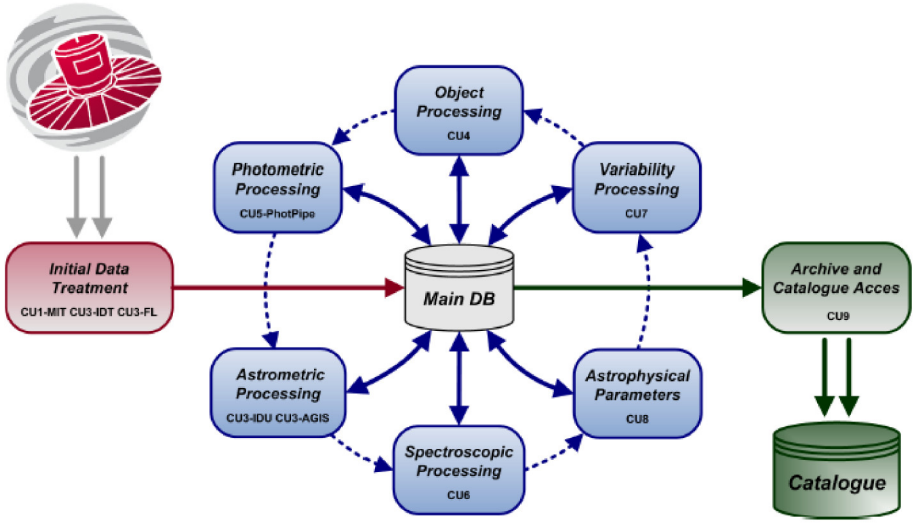

FIGURE 2.4: Gaia DPAC processing cycle, from raw data to the final catalogue. (Credit: IEEC-UB)

to know if the hardware that will be used until the end of the project will be the same or if it will change. Thus, it is necessary to be able to easily migrate the code from one platform to another.

2.3 Intermediate Data Updating

In Gaia there are nine Coordination Unit (CU) responsible for producing the scientific software that is executed in the DPCs. For this thesis, CU3, in charge of the core astrometric processing tasks, is the most relevant one. One of the core tasks within CU3 is IDU. This system is responsible to update, according to the latest calibrations, all the intermediate data including fluxes, locations or cross-matching of observations to sources.

IDU must treat in each cycle all the accumulated astrometric data since the beginning of the mission. It is one of the most demanding systems within DPAC. IDU is executed at the MareNostrum Supercomputer, hosted by the BSC. The supercomputer delivers enough processing power to execute all the required processes. However, due to the huge amount of data and the design of the BSC MareNostrum supercomputer architecture, it is not possible to use a database and thus all processing is file-based. Fig. 2.5 shows the main tasks of IDU and the dependencies between them and the data.

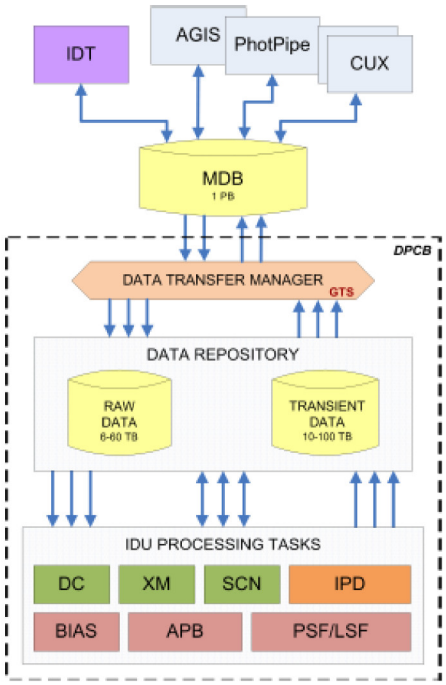

FIGURE 2.5: Overview of IDU processes. (Credit: IEEC-UB)

2.3.1 Cross-Match

One of the most crucial tasks in the Gaia data reduction process is the IDU XM. The goal of the XM is to provide links between the individual detections and the source entries in the catalogue (that is, astronomical sources, mainly stars). In other words, it must decide which catalogue entry each measurement corresponds to. This process is required for downstream processes that will take all the observations of a given source and provide further information or processing refinement on it.

The number of detections that must be handled by this process is huge (in the order of 10^{11}). Therefore, it is not possible or efficient to handle them in a single process. However, the XM is a global process, which has to take into account all the available detections. In order to solve this, the system has been designed in three main steps:

- **Detection processor**: detections do not have position information, only a very accurate value of when the measurement was taken on-board the satellite. This first step takes the observations, determines their sky coordinates and compares them with the list of sources from the current Gaia catalogue. Sources are selected as candidates for an observation according to a distance criterion. The combination of an observation plus the source candidates computed is referred to as

MatchCandidate. The task is distributed in blocks of observations by time. Each block will result in a set of *MatchCandidate* records stored by space (their position in the sky). This is because the next task works in spatial blocks rather than time blocks.

- **Sky partitioner**: this task groups the previous results according to the source candidates provided for each detection. This task is designed to avoid contour problems when processing the data by space. Basically, the goal is to group the related *MatchCandidate* records (by considering shared source candidates) into a single group called *MatchCandidateGroup*. Each *MatchCandidateGroup* contains all spatially-related *MatchCandidate* records. Therefore, the next step can safely process each *MatchCandidateGroup* independently being sure that the handling of a given group will not affect others.

- **Match resolver**: at this point a given observation will most likely have several sources flagged as possible candidates. This final task must resolve all the conflicts and provide a consistent match for each observation. The goal of this process is to correctly identify all the observations of a given source and create new sources if there was no entry on the catalogue for a given observation.

2.4 Data access challenge

The IDU XM is a very complex process. Apart from the actual algorithm complexity, a very challenging aspect of the task is the data organization. As shown in Sect. 2.3.1, the XM process requires different organizations of the data in each step: time-based, space-based or group-based. It is therefore crucial to optimize the data access layer for the task.

However, as already mentioned in Sect. 2.3, it is not possible to use a database to store the data for obvious reasons. Hence, the data in IDU is stored in files and it is important to manage these files efficiently. In order to have less overhead, the number of files should be minimized. The size of the file is also important, as having too large files requires more memory to process them, but having too small files will result in inefficient I/O operations and extra overheads. This is specially important in the Gaia data processing, and in particular for the IDU XM task. The number of elements that enter this particular

task is huge, but each record is rather small in terms of size. The BSC MareNostrum file system has a rather large block size (currently 1MB). This means that any file will take 1MB from the file system, even if the actual data is smaller. Therefore, an important equalization task before running IDU is to distribute the data (and therefore the jobs) into blocks of the adequate size in order to be executed efficiently in the cluster.

The optimization of the data access layer is the main scope of this project. In particular, the aim is to optimize the DAL in order to find a efficient way to save the huge amount of data coming from the spacecraft and to retrieve them in a efficient and smart way.

2.5 The Gaia Binary file format

DPAC established a common file format known as Gaia Binary File (GBIN). It was designed mainly for data exchange between the different DPCs and not really to be used in data processing, because most of the DPCs use databases as they process smaller sets of data at one time.

GBIN files consist of a header describing the data type, version and basic information. Then, blocks of serialized Java objects are included, compressed with the Zip library. This design is not optimized for accessing random parts of the file efficiently. The contents of the file must be all read and then filtered in order to find a specific part of the data, as they are not indexed or directly accessible inside the GBIN file. Fig. 2.6 shows the general structure of a GBIN file.

In general, the file is composed of one header and one or more data sections. It is possible to add data also once the file has been already created, appending one additional section to the already existing ones. The header is used to maintain meta-data and general information regarding the identification file, file header, format version number, etc.

The file header is followed by the multiple data sections, each composed of a section header, serialized data and the end of section marker. The data is, simply, the serialized Java objects, converted into a byte array. The serialization algorithm is quite simple and straightforward in order to facilitate the usage at all the various DPCs.

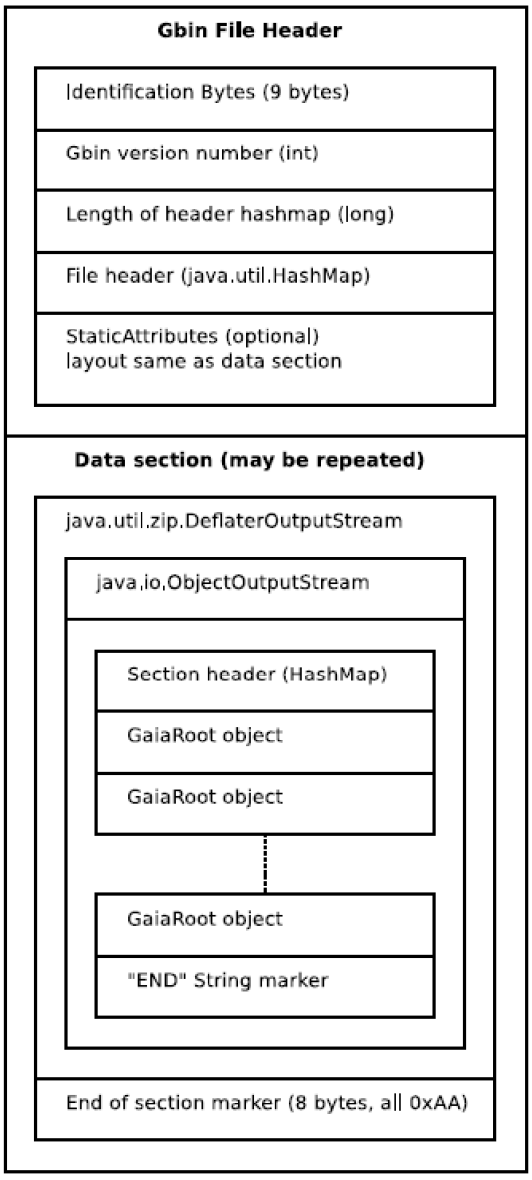

FIGURE 2.6: Gaia Binary File structure. (Credit: DPAC)

Chapter 3

HEALPix sphere tesselation

The HEALPix is an adaptable structure for the pixelization of the sphere. The library was developed for the cosmic microwave background experiments (i.e., BOOMERANG [9], WMAP [10]) in the 1990s. In fact, it was designed as an efficient library for the data processing but especially for the data analysis. Nowadays, it is also used to handle the new challenges raised from new experiments regarding discretization with associated hierarchical indexation and fast analysis or synthesis of functions defined on the sphere [11]. In particular, in projects with large amounts of astronomical data, it is possible to create discretized spherical maps and work directly on them. This allows the creation of scientific applications that are able to handle the data more efficiently by using the HEALPix library.

The library includes very efficient computational algorithms. The framework also includes software tools supporting the visualization of the data [12]. The library has been implemented in C, C++, Fortran90, IDL/GDL, Java and Python, each with its corresponding documentation.

The creation of pixelized sky maps is one of the most important stages in the data processing of such projects. This distribution allows to store the data only at the required resolution level, reducing the amount of data by keeping just the values for the physical parameters of interest at the relevant resolution level. The discretization also allows for data equalization (either in terms of volume, records, etc.) and therefore easier and better distribution of the processing loads.

The relevance of these pixelization methods is increasing as each new detector is able to get more information (often orders of magnitude beyond) than previous generations. This situation is particularly relevant for scientific space missions and projects that perform measurements distributed among the whole sky. In these situations it is complicated to perform the data reduction and the scientific extraction of information if no pixelization is used, as the data volume is too large to be processed at once or sequentially. As a reference, in the 1990s the usual number of pixels with measurements in these projects were around 6000, whereas nowadays they are around 50 million.

In order to retrieve the correct and relevant information from these large amounts of data, in each processing task it is necessary to:

- Resolve the problem with a global analysis (spatial correlations, harmonic decomposition, etc.)
- Analyze the space morphology, especially regarding the detection, identification and characterization of the different objects in the spatial domain.
- Allow the correlation with external data sets (useful for an in-depth study of the spatial retrieved data).

It is not possible to solve these problems unless there is a powerful framework that allows a precise definition of the data model [11]. In particular, the precious observations coming from spacecrafts are not always easy to be interpreted. Thus, it is necessary to have a really efficient and well developed framework that it is able to analyze the resources in a proper way through reliable mathematical algorithms. At the same time, one should also consider time constraints. Analyzing these kind of resources could require a really long time. Therefore, the mentioned framework should be able to do it in a reasonable amount of time. It is possible by taking care of the data model and defining it in a efficient manner [11].

The HEALPix framework is especially useful for the definition of digitized sky maps, which is a nearly mandatory stage or processing step between the large amount of data retrieved and the last stage of astrophysical data analysis (or data reduction) [11].

Several solutions were tested to develop the HEALPix library: Quadrilateralized Spherical Cube, Equidistant Cylindrical Projection, Igloo-type, etc. This was done especially

for the mathematical structure of digitized whole-sky maps. Several features were evaluated for the performance of the framework. Firstly, a hierarchical structure of the database was preferred in order to facilitate some topological methods of analysis, especially for triangle and quadrilateral grids.

Secondly, it was considered important to have equal areas of discrete elements of partition. In fact, this helps to integrate the white noise in the pixel space and to avoid regional dependencies. Finally, but not less important, an iso-Latitude distribution of discrete area elements on a sphere was preferred. Thanks to this, it is possible to speed up the time, especially when the computation time is the main limit of the spherical harmonics. More details are available in [11].

3.1 Hierarchical indexing

An important feature to take into account when data distributed on a sphere is processed (or more specifically, astronomical data) is the possibility of referring to it in a hierarchical way. The hierarchical organization of data allows to optimize several methods of analysis, especially when the amount of information is reduced from previous processes with specific methods (digitized sky maps could be an example).

In fact, it makes possible to handle the data in a easier way since it is more organized than during its acquisition. Moreover, it allows a more direct access to specific parts. Actually, with a proper data model and file format, it is possible to access only the data that is strictly needed for a given process.

The HEALPix library allows the distribution in quadrilaterals of equal area, but with different shapes. In particular, everything is based on the lowest-level (or base resolution) sphere tesselation, namely, twelve pixels in three rings around the poles and equator [13], as illustrated in the top left case of Fig. 3.5 hereafter.

It is possible to have two different numbering schemes for the pixels: *Ring* scheme, and *Nested* scheme. In the Ring numbering scheme, the pixels can be counted from the north to the south pole along each iso-latitude ring. Figs. 3.1 and 3.2 show the Ring numbering scheme, with parameter $N_{side} = 2$ and $N_{side} = 4$ respectively, where N_{side} is the resolution of the pixels. This pixel resolution defines the number of divisions

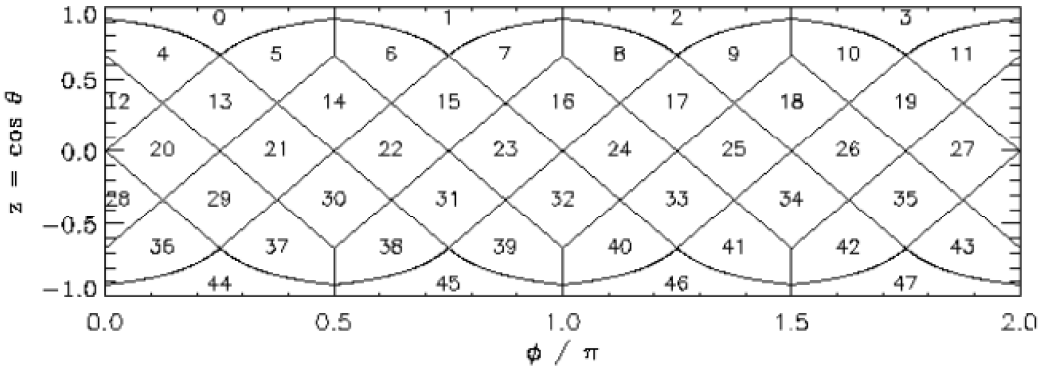

FIGURE 3.1: Example of Ring scheme with $N_{side} = 2$.

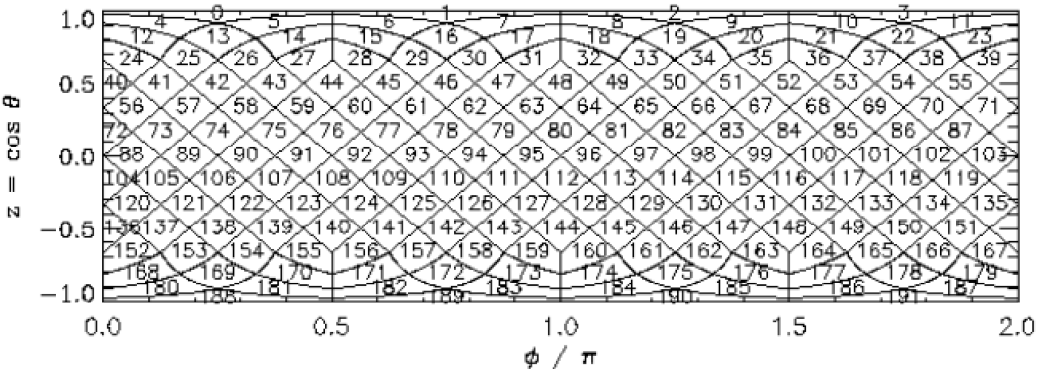

FIGURE 3.2: Example of Ring scheme with $N_{side} = 4$.

along the side of a base-resolution pixel that is needed to reach a given high-resolution partition [13]. Note how the pixel numbering goes down from north to south pole through consecutive iso-latitude rings.

In the Nested numbering scheme, the pixels can be indexed in twelve tree structures, corresponding to base-resolution pixels, as shown in Figs. 3.3 and 3.4. As in Figs. 3.1 and 3.2, the two examples of these figures show the numbering with $N_{side} = 2$ and $N_{side} = 4$ respectively. Note how, in this Nested scheme, the pixel numbering grows with consecutive hierarchical subdivisions on a tree structure seeded by the twelve base-resolution pixels. The pixel numbering is clearly different from the Ring scheme shown before.

It is important to highlight that the numbering scheme, specially the Nested one, can be easily implemented and allows an efficient implementation of nearest-neighbor searches [13].

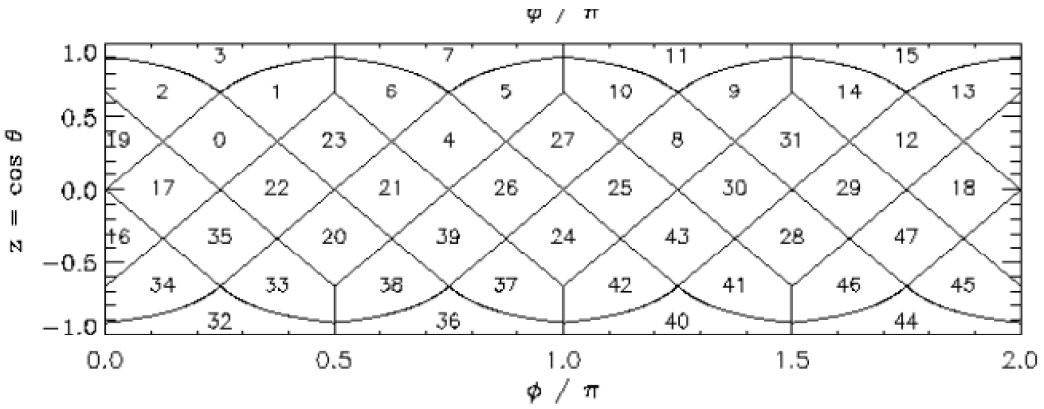

FIGURE 3.3: Example of Nested scheme with $N_{side} = 2$.

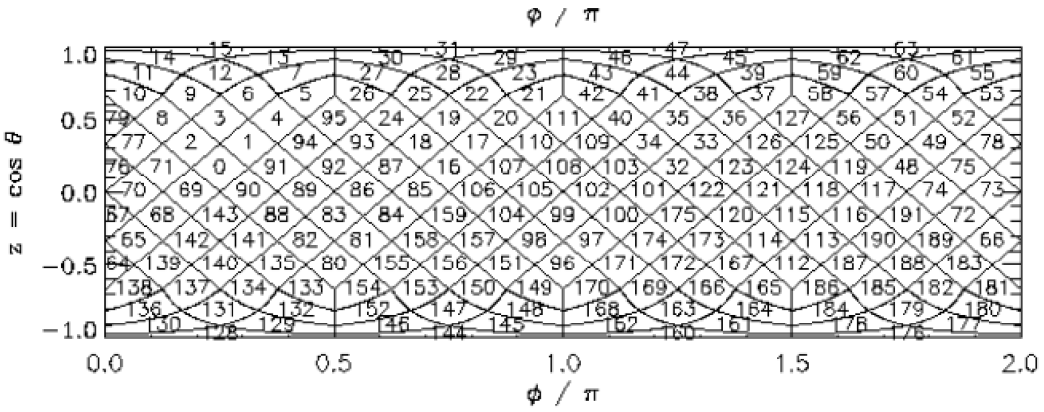

FIGURE 3.4: Example of Nested scheme with $N_{side} = 4$.

3.2 Structure and components

According to the way in which the spherical area is pixelated, each pixel contains the same surface area. As anticipated before, the base-resolution comprises twelve pixels in three rings around the poles and equator. That is, the entire spherical surface is hierarchically divided into curvilinear quadrilaterals, and the lowest resolution level is composed of twelve base pixels. We refer to the lowest resolution level as *level 0*.

Each pixel may then be divided into four new pixels (of the next level). This can be repeated until the highest level (highest spatial resolution) is reached. In the current HEALPix library the maximum level is 12. Obviously, each time that one pixel is split, the area covered by the respective *sub-pixels* becomes proportionally smaller. However, the area covered by each pixel of the same level is always the same.

All iso-latitude rings located between the upper and lower corners of the equatorial base-resolution pixels, the equatorial zone, are divided into the same number of pixels:

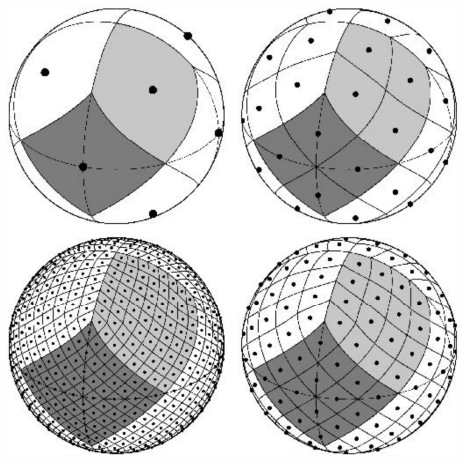

FIGURE 3.5: Tesselation of the spherical surface from level 0 to 3 (in clockwise order).

$N = 4 \times N_{side}$. The remaining rings, located among the polar cap regions, contain a variable number of pixels, increasing from ring to ring as the distance from the pole increases.

The linear and constant latitude distribution of the pixels is a fundamental property of HEALPix. This feature allows to compute the data in a faster way. With this, it is possible to have a faster discretization of the spherical harmonic transforms.

Fig. 3.5 shows the division of the sphere down to HEALPix level 3. As detailed in Sect. 3.1, the base partitioning has twelve equally sized pixels in three rings named as follows:

- N (North): N0, N1, N2, N3
- S (South): S0, S1, S2, S3
- E (Equator): E0, E1, E2, E3

From these base level pixels it is possible to divide again each pixel in four smaller pixels. These can be created in a recursive way to obtain level 1 pixels. Applying this process we can further divide each pixel until the maximum HEALPix level, 12. Each new level is coded adding one digit between 0 and 3 to the previous base pixel, and each time that one pixel is split, also the surface is proportionally four times smaller, providing more spatial resolution [14]. For instance, level 1 sub-pixels of base pixels N0, S0 and E0 are the following:

- N0: N00, N01, N02, N03
- S0: S00, S01, S02, S03
- E0: E00, E01, E02, E03

For this project, these methods of hierarchical coding are useful to refer to pixels, assign group names, etc. The pixel name will be used to retrieve the information in a convenient and efficient way.

This hierarchical naming for the pixels has the disadvantage to be a string value, taking up a significant data volume when storing the data. The HEALPix library also provides an equivalent numeric identifier for the pixels (an integer value). However, numeric identifiers have no hierarchical organization, which might require additional operations to compute the affected pixels in some data queries. Chapter 5 contains all the implementation details.

3.3 Usage in Gaia/DPAC

The analysis of spherical topologies is complex, especially when data must be accessed in a spatial manner. In fact, in some fields such as astronomy or geophysics, the geometry is not linear and could be defined either from the objects that have to be studied or by the need to use some particular techniques [11].

In the particular case of the Gaia mission, as the satellite scans the whole sky, and due to the huge volume of data, some partitioning is required for the data processing. In the Gaia DPAC, the HEALPix library is used to partition the data in a fast and efficient way, specially for tasks that must process all the accumulated data, such as IDU.

Specifically for this project, the HEALPix library is also very convenient as it allows an easy coding of the different levels of the spherical surface. This allows for a convenient naming of the different groups and data sets in the *HDF5* file structure (see Chapter 3 hereafter). This hierarchical structure is used to locate data easily within the file.

Chapter 4

The HDF5 file format

Hierarchical Data Format (HDF) is a powerful set of libraries that allows users to save, organize and manage data when it is not possible, or it is restrictive, to use traditional database systems. It was developed for the first time by the National Center for Supercomputing Applications (NCSA) in 1987. Today it is supported by the HDF Group, a no-profit corporation located in Champaign Illinois at the University of Illinois, whose aim is to support and improve the performance of the product with the changing technologies, maintaining always a high level of quality and reliability [15]. Thanks to a continuous development, they ensure a long-term access to the HDF stored data. In particular, there are several features that make the technology interesting:

- It is portable, allowing to share data between different computational platforms written in different languages.
- Allows very large and complex datasets, accessible in a very fast way and provides storage space optimization.
- Objects can be related using complex data relationships and dependencies.
- It is possible to refer to hierarchical data objects in a very natural manner.
- Each dataset can be represented in an n-dimensional manner, where each element may be a complex object.
- It allows to process all records sequentially in a efficient way but also the usage of coordinate-style lookup.

- It is possible to use several tools and applications for managing, manipulating, viewing, and analyzing the data.

In addition, HDF is open-source and its distribution is totally free, which makes it a good choice for projects that must export data to the global community.

HDF files and objects are created, accessed and manipulated using the APIs, or several interfaces available with the HDF library. The library is implemented in C, but the functions are also wrapped in other languages to facilitate the work with them [16]. However, currently the Java interface has some limitations and therefore in this project we access the C API from Java through JNI.

4.1 File operations

An HDF5 file is created using the `H5Fcreate` function. It is possible to control the behavior if the file already exists on the file system with the following flags:

- `H5F_ACC_TRUNC`: if the file already exists, all the data on it will be overwritten.
- `H5F_ACC_EXCL`: if the file already exists, the creation of the file will not be allowed.

There are two additional parameters required in order to create the file: the *creation properties list* and the *access properties list*. The first one is used to store the configuration about the version that is used and to set parameters of global data structures. The second parameter is used to set parameters for garbage collection, caching and for access parameters. Once the *creation properties list* are set it is not possible to change them. However, the *access properties list* can be modified when the user closes and reopens the file. Default values can be used for the properties.

Existing files can be accessed through the corresponding `H5FOpen` function. The file can be opened in two different modes:

- `H5F_ACC_RDONLY`: it is only possible to read from the file, but not to write on it.
- `H5F_ACC_RDWR`: it is possible to both read and write on the file.

HDF5 allows opening a file more than once. This means that concurrent access to the file is allowed, but at the same time, different identifiers might actually refer to the same physical file. When the file is not needed anymore all the identifiers should be released by closing the file. In fact, each object that has been opened must be closed when it is not required anymore.

Both `H5Fcreate` and `H5FOpen` return a unique identifier for the file. Further operations on the file are executed by providing this unique *file ID*.

4.2 Structure

HDF files can be seen as a container composed of two primary types of objects, organized in a hierarchical way as a direct graph and with an unique root named "`/`". The two main objects are:

- **Groups:** they contain other groups or datasets. These are described in further detail in Sect. 4.3, and can be seen as the equivalent of directories in a UNIX file structure.
- **Datasets:** they contain data written in a specific format which is stored in the dataset itself. More information about datasets can be found in Sect. 4.4.

Fig. 4.1 shows a simple example of an HDF5 file structure with groups and datasets.

It is possible to define other types of objects but every object is contained in the root group. They will be discussed later in this chapter. Each node represents an object in the file and they are connected to each other through links. It is possible to refer to the objects with an absolute path (starting at the root group) or a relative path (starting at one specific group). The paths are basically a string of components separated by "`/`". All the objects in one file can be discovered traversing the graph in a recursive way starting from a node, that could be the root or an intermediate one. This is possible because the file is self-described.

It is reasonable to see an HDF5 file as a UNIX file system. Objects are stored on it and they can be accessed through paths (or links). A single data item can also be referred by more than one link. This allows to store objects in any memory position, and still access

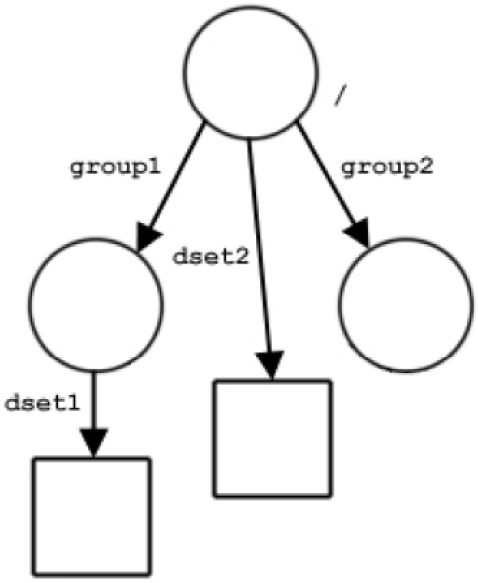

FIGURE 4.1: Example of an HDF5 file structure with groups and datasets

the data directly. Furthermore if a link is removed, modified or added, the position of the actual objects in the file does not have to change. Thanks to this linking mechanism, it is possible to build complex structures, including cyclic references in the file.

4.3 Groups

HDF5 groups can be seen as directories in a Unix file system. They contain objects (or they can also be empty), other groups or datasets. There is only one object that is not a member of any group, that is, the root. The relations between groups are implemented via link objects. In particular, each link has a name, and each link points to exactly one object. However, the objects can be referred with more than one link, but the object identifier has to be unique in the group. The name of the link represents the path name of the group. In fact, it is composed by strings where each one corresponds to one object in the file. The several groups are divided by a slash, `/`.

For instance, the link `GroupA/GroupB/Dataset1` points to `Dataset1` in `GroupB`, contained in `GroupA`. Hence, what the links represent in the file is the hierarchy between the objects. Taking advantage of this method, it is possible to create very complex data structures, using many levels of hierarchy. A programming model should be followed in order to work with groups in HDF5:

1. Creating or opening the group, obtaining a unique group identifier.
2. Performing the necessary operations on the group, via the identifier.
3. Closing the group, releasing the unique identifier.

Groups also have *creation properties list* and *access properties list* for the corresponding `H5Gcreate` and `H5Gopen`. Both functions return a unique identifier for the group. Once a group identifier is available it is possible to create a new object in the group, delete an existing object, iterate over the available objects, etc.

Once the operations with the group have finished the group has to be closed, releasing the identifier, as well as all the objects opened inside it, in order not to lose integrity [16].

4.4 Datasets

Data in HDF5 files is stored in multidimensional arrays that are described by the dataspace object. The single data element (basically a set of bits) can be described in different ways thanks to the different datatypes, see Sect. 4.6. Basically, a dataset object is stored in two different parts:

- Header: it contains the metadata. The size is variable but, typically, when added to a simple dataset with a simple datatype, it takes about 256 bytes. However, the size depends on the information that is stored in the header regarding datatype, filters and layout of the dataset among other information [16]. The header might also contain user-defined attributes that can be added to the object. See Sect. 4.7 for more details.
- Data array: the raw data that is stored on the dataset. The size depends on the strategy used to allocate and represent the data.

Similar to all the other objects described in previous sections, datasets follow the same programming model:

- Creation/opening of the dataset.
- Performing the needed operations on it through the unique dataset identifier.

- Releasing the dataset.

Similarly to the group and file operations, the creation of the dataset is possible using the function `H5Dopen`. In this case some information must be provided:

- Name: a sequence of alphanumeric ASCII characters.
- Dataspace: it describes the space in which the dataset will be stored in terms of number of dimensions and size of each dimension.
- Datatype: it specifies how the data must be interpreted.
- Storage properties: they describe how the data is organized inside the file.
- Filters: they allow setting additional features to the dataset such as compression or error detection.

One of the basic operations over a dataset is to write data on it. In HDF5 it is possible to use different layouts in order to store the data:

- Contiguous: it is the simplest model, where the data is stored contiguously as a unique block. There are some limitations that have to be taken into account, such as fixed-size (once the dimension is set, it is not possible to change it) and some filters that cannot be used, such as compression.
- Chunked: the data is divided and stored in smaller fixed-size chunks. The size of the chunk can be specified by the user and it is stored in the dataset. Each chunk will be read or written with only one operation.
- Compact: it is particularly useful when the dataset to be saved is pretty small. This is because, as said before, the dataset is stored in two different parts (header and data). Unavoidably, two operations are needed to read the data, one for each part of the dataset. This is not very efficient if the dataset is small. In this type of dataset it is possible to read both parts with just one operation instead of two. The dataset should be less than 30KB.

The creation of the dataset returns a unique identifier for the dataset [16]. The API provides two basic functions that operate on this unique dataset identifier and allows the

user to read and write data: `H5Dread` and `H5Dwrite`. Once all operations are executed and the dataset is not needed anymore, the resource has to be released. This is done with the `H5Dclose` function.

4.5 Dataspaces

Dataspaces define the size and shape of the raw data and possible attributes that could be added to the objects. In particular, they create the region of space in which the data will be saved in terms of number of dimensions and size of each one.

The dataspace has to be created when the dataset or the attribute is created. However, this is not the only role of the dataspace. It is actually also used for I/O operations, especially when data has to be transferred from a file to memory or vice versa.

In general, through the API functions it is possible to manage the dataset, retrieve information of it and execute selection operations [16]. There are three different kinds of dataspaces:

- Scalar: the dataspace is composed of just one element, although this could be a complex data structure.
- Simple: a multidimensional array of elements. The dimension of the array could vary from the current size to the maximum one. The maximum size can be unlimited (bounded to the biggest integer value available on the system). The dimensionality has to be declared at creation time.
- Null: no data elements are defined inside, hence, no selection is possible in this kind of dataset.

It is not necessary to read or transfer all the data together. It is possible to read only one part of the dataset, or just a few elements from it. The elements can be selected in two different ways:

- Hyperslab: selecting elements from a hyper rectangle.
- Point: array of points as coordinates.

Further details are beyond the scope of this project, because here all the dataspaces to be considered are *simple* and data is read and written with only one I/O operation.

4.6 Datatypes

The HDF5 datatypes define the layout of the data and how it will be stored in the dataset. The datatype must be defined when a dataset or an attribute is created. Essentially, it specifies how to interpret the data when it has to be stored or transferred between two different regions of memory. Once the dataset or the attribute is defined, the datatype cannot be modified.

It is interesting to consider that HDF5 allows the conversion between some datatypes. For example, Integer to Integer, but one can be little-endian and the other big-endian. The data will be converted transparently by the library.

The HDF5 library has defined 11 classes of datatypes divided in:

- Atomic: Byte, Bitfield, String, Time, Reference, Integer, Opaque, Float.
- Composed: Array, Enumeration, Variable Length, Compound.

The main difference between them is how they are defined. Particularly, regarding the composed datatypes, they can also be structures of one or more datatypes. Therefore, complex structures with several levels can be created. This flexibility allows the user to implement the datatype that is better adaptable to the data to be stored. The information regarding the datatype will also be stored as metadata on the dataset or attribute when this is created.

Due to the nature of the data in this project, particular attention should be paid to the *opaque* datatype. This kind of type is used in order to store data that is left *uninterpreted*. The reason behind this is to provide complete flexibility to store any of the very complex and extensive list of datatypes used in the Gaia data processing. Java objects are serialized and converted into a byte array, which is seen by HDF5 as an *opaque* datatype. Upon creation, the dimension is set according to the corresponding data size.

4.7 Attributes

Attributes are used when it is necessary to keep additional information attached to a given HDF5 object. These can be added to any primitive object (such as file, group or dataset). It is not mandatory to have an attribute when the item is created, and it is possible to have multiple attributes attached to a given object.

Generally, attributes are rather small in terms of size. In general, if data is larger than 64KB it is recommended to use a normal dataset to store it. The user is allowed to define the datatype of the attributes, although some limitations apply:

- It is not possible to compress or to chunk the data (since the data is supposed to be small).
- The information has to be read altogether, with only one I/O instruction.
- They cannot be shared and it is not possible to add attributes on attributes.
- It is attached on the object header of the object being described.

Functions `H5Acreate` and `H5Aopen` allow the user to create or open attributes and return a unique identifier. This identifier is needed to refer or use the attribute in the application. When it is not necessary anymore, it can be closed or deleted altogether.

4.8 Applicability to Gaia/DPAC

HDF5 has been considered due to the IDU processing requirements and the limitations imposed by the hardware where it must be deployed. As previously said, since in MareNostrum there is no central database, access to the data has to be done through files. However, it is clear that it is not recommended to access a huge and very complex directory structure with relatively small files in a shared file system (such as General Parallel File System (GPFS) in BSC-MareNostrum), basically because the I/O performance will be affected negatively. Furthermore, the quantity of data that is going to be saved is really huge, specially as it is always increasing in each processing cycle, as explained in Chapter 2. In fact, in each cycle new data is added to the already accumulated one, and everything must be processed together in the next IDU execution.

The number of records to be processed is as important as the data volume. Data should be split in different files in order to allow tasks to be easily distributed in the computing nodes. However, the granularity of this partition will have to be assessed according to the performance of the I/O operations. It is necessary to find a trade-off between the quantity of data stored in a given file (and thus the memory required to read it) and the performance of I/O operations. Smaller files provide greater granularity and require less time to read and process once loaded. However, there is a penalty when creating and accessing huge numbers of small files in the file system.

At the same time, it is not very convenient to use the generic file format, GBIN, used to exchange and store data in DPAC either. Not only because GBIN is a particular file format whose structure is not adapted for data processing, but specially because it only allows to access the data sequentially without providing any structure within the file.

As detailed in Sect. 2.3, it is necessary to retrieve data objects from the files according to the time criterion for some given tasks, and according to a space criterion (in terms of distribution on the sky) in others. GBIN files cannot provide any of the required data organization within the file. In order to arrange data according to these criteria, offline data arrangement processes have to sequentially read all the data and slowly rearrange it to the desired partition. This is very inefficient in terms of computing time required and it is also very difficult to scale or distribute.

HDF5 provides an easy way to implement hierarchical file structures. At the same time, HEALPix provides a convenient hierarchical partition of the sphere, which allows an organized division of the data, as seen in Chapter 3. Hence, it is natural to use the same strategy to store the data in a hierarchical way, creating several groups according to the HEALPix partition scheme. The hierarchy can be different depending on whether the HEALPix index number or the HEALPix name is used, but in any case the information will have a clear structure. This data distribution within the file allows easy and direct access to the different parts of the data, thus allowing to have fewer files and a more efficient I/O.

As explained Sect. 4.6, in HDF5 it is possible to specify the type of the data that is going to be stored in the file. However, it is also possible to configure HDF5 to avoid interpreting the data. This is exactly the most suitable configuration in this case. This is due to the fact that there are a lot of different and complex datatypes used within

DPAC, so implementing and maintaining specific HDF5 datatypes for each of them would be a huge effort. Moreover, the data is actually processed in Java, so it would have to be converted back into a Java object which would be very inefficient. By using opaque datasets with serialized java objects the file format is directly applicable to all existing DPAC datatypes. At the same time, data is treated in a more consistent way, decreasing also the quantity of code that should be implemented.

Taking into account all these restrictions and considerations, and knowing that the main advantage of using HDF5 is that a file is self-described, binary data can be included in the files and direct access to the different parts of a file is possible — without first having to parse the entire file. Also, there is no limit to the quantity of data that can be stored on the files, and conveniently, metadata can also be stored with the file. The conclusion is that the HDF file format can be a great solution for this project. In this thesis we particularly use the HDF5 library, version 5-1.8.16.

Chapter 5

Implementation

The aim of the project is twofold: the definition of an efficient data storage structure for Healpix-based data in HDF5 and the definition and implementation of a JNI interface to communicate with Java. Moreover, a pre-processing stage should be implemented in order to allow data to be sent through the defined JNI. After this, the I/O performance shall be analyzed. If as a result of the performance analysis, optimization actions are required these shall be implemented as long as they fall within the scope of the project.

For the first part of the project we have devised two different solutions: *Full hierarchical HEALPix structure* and *Flat HEALPix structure*. The difference between them is how the data will be stored on the file, which is translated into HDF5 groups and dataset structure. Sections 5.1 and 5.2 provide detailed descriptions of both implementations. The implementation has be done in C, using the HDF5 library, version 5-1.8.16. Special attention was given to write and read data in the most efficient way. The development was split into two natural blocks, one with the write features and another for the read routines. Some shared routines were included in a common library.

5.1 Full hierarchical HEALPix structure

In this implementation the HDF5 file is organized as a hierarchical tree structure (or direct graph) following the HEALPix structure. It is also possible to see this structure as a UNIX file system where the UNIX directories are mapped to HDF5 groups and the

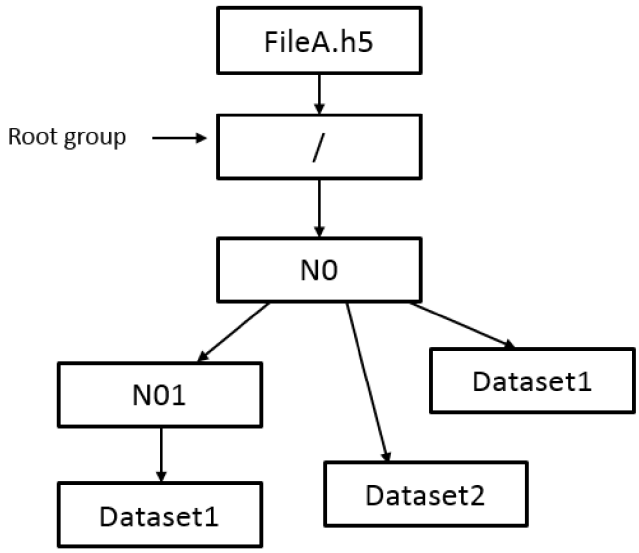

FIGURE 5.1: Example of a file following the hierarchical HEALPix structure

files to the HDF5 datasets. Fig. 5.1 shows a simplified example of the structure used in this implementation.

The file is composed of groups and datasets and it is possible to refer to them using links [16]. In particular, a path name (link) is a string of components separated by forward slashes `/`. The forward slash specifies the relations between the different groups in a file, specially regarding their hierarchical relationship: the object following the slash, is included in the object preceding the slash. Each path can start with the character dot (`.`) to indicate the current group, the character slash (`/`) to indicate the root group or any member of the current group in case of relative paths.

Following the example shown in Fig. 5.1 we could access the groups through full or relative paths:

- `/N0/Dataset1`: full path to `Dataset1`.
- `/N0/N01/Dataset1`: full path to `Dataset1`
- `Dataset1`: relative path, from `N0`, to Dataset1
- `N01/Dataset1`: relative path, from `N0`, to Dataset1

Datasets are identified by their name according to the different levels. For example, having a name as N012012 means that it contains HEALPix data at level 5 for this specific pixel. In order to store the data in the right hierarchical location within the file, all the intermediate groups must be created before writing data to a new dataset. Following the example shown in Fig. 5.1, the dataset will be created in the group `N0/N01/N012/N0120/N01201/N012012`. If at the time of writing this data any of the intermediate groups are missing they will have to be created.

It is not possible to have multiple objects with the same name inside a given group [16]. Theoretically, there is no limitation for the number of datasets and groups in a HDF5 file.

The HDF5 format is very convenient to keep the information in the same hierarchical structure in which it is defined. Moreover, data retrieval is straightforward as only the pixel name is required to retrieve it.

5.1.1 Writing

In order to write the data, the function `WriteHdf5File` has been implemented. Basically, the function receives the HDF5 identifier of the file, the data to be stored as a byte array, the HEALPix name and level.

Writing contents to the HDF5 file is done in four basic steps. First the full HEALPix name is built in order to save the data in the corresponding group. This is done through two functions which build the full path. The path starts from the root `/` with all the hierarchy until the requested level. For example, if we take N0321 then the function would create N0/N03/N032/N0321 as full path. In this example the level required is the 3^{rd} one and the created path will allow to create the target group.

Once the full path of the group is created, it has to be checked if the required group already exists or not in the file. If the group does not exist all or some of the intermediate groups may not have been created yet. Thus, it is necessary to create them before writing the data into a Dataset. Following the previous example, it is fundamental to create the groups N0, N03 and N032 before to create the group N0321. This is done by requesting HDF5 to create all the intermediate groups through a property list. Once the desired group structure exists, the dataset can be created.

Before data can be stored in the corresponding dataset we must determine the next dataset name. Section 5.3 provides an in-depth discussion on the naming and tracking of the number of available datasets. Once the data set name is available, the function `HDF5_Create_Opaque_Dataset` is called which creates the data set and writes the byte array into it.

The final step is to correctly close all the HDF5 resources in order to avoid loss of integrity in the file.

The following programming model must be followed to write data to the file:

- Create the full path of the group where the data will be saved on.
- Create the group in the file, if it does not exist yet.
- Create the opaque dataset calling the function `HDF5_Create_Opaque_Dataset`, which will also store the data into it.
- Release all the resources and free previously allocated memory.

5.1.2 Reading

In order to read the data, the function `ReadHdf5Dataset` has been implemented. The function receives as input the file path and the absolute path of the dataset to be retrieved.

Reading contents from the HDF5 file is a two step process. First, from the Java application the user requests the list of datasets that contain information for a given HEALPix in the current file. The request returns a list of absolute, or full, HDF5 dataset paths. Once the full dataset paths to be read are available the read operation is very simple. The user can directly call the `ReadHdf5Dataset` function and provide the full data set path. The HDF5 code only opens the group, reads the opaque dataset and delivers back the contents to the Java application through JNI.

The following programming model should be followed in order to read information from the HDF5 file:

- Retrieve the list of datasets containing information related to the required HEALPix in the file.

- Call the function `ReadHdf5Dataset` in order to read the data.
- Return the information to the Java side.
- Release all the resources and free previously allocated memory.

5.2 Flat HEALPix structure

A second solution was proposed to store and retrieve the data as efficiently as possible. In this alternate solution there is no intermediate HEALPix group structure and all groups are at the HDF5 root level. This reduces the number of operations required to store the data in the appropriate location within the file.

For instance, it is not necessary to build the full dataset path to create, open or check if an object already exists or not. At the same time, all the intermediate groups are avoided and there is always only one group at the root level which contains the datasets with the actual data. This avoids the creation of potentially a lot of intermediate groups when storing data at a very high HEALPix level.The idea behind this is that having less intermediate groups could decrease the dataset creation and access time.

All the groups are identified by a unique HEALPix numerical identifier. In this implementation groups are named according to this *numerical* string which identifies a particular HEALPix given the level at which it has been computed [13].

In particular, it is possible to use three parameters in order to recognize a pixel:

- Numbering scheme: RING or NESTED
- Resolution or size parameter: N_{size}
- pixel p range between $[0, 12(N_{size})^2 - 1]$

It is really useful to refer to a pixel in a unique way using the pixel range $[0, 12(N_{size})^2-1]$, where $N_{side} = 2^k$. K is the level at which the data are considered. Thus, the pixel range for the project will be $[0, 201326591]$, as 12 is the highest level.

5.2.1 Writing

The function `writeHdf5FileFlat` has been implemented in order to write the data in the right position in the file when using the *flat* implementation. In particular, the function receives as input parameters the HDF5 file identifier, the buffer with the data to be stored, the HEALPix identifier and the level.

The information in the HDF5 file is written executing three basic steps. First of all, as in the hierarchical case, it has to be check if the group already exists. If the group does not exist it has to be created. However, in this case it is not necessary to create any intermediate groups the structure is *flat*. Furthermore, the full path is just the HEALPix identifier which the function receives. In order to know always at which level the information is written it is necessary to store this information as an attribute of the group. Thus, the attribute `LEVEL_ATT` is created and written into the group. If the group already exists this step will be skipped.

Also in this implementation before storing the data in the corresponding dataset we must first determine the name of the next dataset. This is done following the same routine used for the *hierarchical* structure as described in Section 5.1.1. Further details regarding the dataset numbering is available in Section 5.3. Once the name of the dataset is known, the function `HDF5_Create_Opaque_Dataset` is called to create the data set and store the corresponding byte array into it. The final step is to correctly close all the HDF5 resources in order to avoid loss of integrity in the file.

This is the programming model that should be followed to write data into the file:

- Create the group in the file with the relative attribute `LEVEL_ATT` if it does not exist yet.
- Create the opaque dataset using the function `HDF5_Create_Opaque_Dataset`, which will also store the data into it.
- Release all the resources and free previously allocated memory.

5.2.2 Reading

In order to read the data, the same generic function implemented and used for the *hierarchical* file structure will be used: `ReadHdf5Dataset`. The function receives as input the file path and the absolute path of the dataset to be retrieved.

In order to read the data from the HDF5 file, two steps are necessary. The first one is to request the list of datasets contained in the file which have information for the requested HEALPix.

Just as in the case of the *hierarchical* structure, we created a function to retrieve the paths of the datasets given a desired HEALPix. The function scans all the datasets and compares the HEALPix level stored in the `LEVEL_ATT` attribute attached to each group with the level of the required HEALPix.If the level is the same, the group identifier is compared and if they match the dataset will be included in the list. If the level is different, it will convert both levels to the one with less resolution and compare them. Once at the same level they can be compared and if they match the full path of the dataset will be added to the list of datasets with data for the requested HEALPix. Otherwise, the dataset will just be skipped.

The second step starts once the full list of dataset paths that contain data for the desired HEALPix is available. Then the user can directly call the `ReadHdf5Dataset` function providing the full dataset path and the HDF5 code will open the group, read the opaque dataset and deliver back the contents to the Java application through JNI.

The programming model is therefore equivalent to the hierarchical one:

- Retrieve the list of datasets containing information related to the required HEALPix in the file.
- Call the function `ReadHdf5Dataset` in order to read the data.
- Return the information to the Java side.
- Release all the resources and free previously allocated memory.

5.3 Dataset numbering

Every time that a new dataset has to be added into a given group we must create a unique name for the dataset as no two elements inside an HDF5 group can share the same name. Therefore a straightforward solution to this problem is to name the data sets sequentially within a group. Obviously to do so we must keep track of the number of datasets already in the group.

There are two alternatives in order to know how many datasets are already present in a given group. The most obvious solution is to simply *count* the number of existing datasets each time that a new one has to be created. This is done by using the HDF5 routines to traverse the group elements. However, HDF5 provides a more elegant solution to store this information by using attributes. In particular the number of data sets in a given group might be stored in an attribute attached to the group. In this case when creating a new dataset there is no recursion needed as only the value must be read and updated.

In this project we implemented both solutions in order to test them and determine the most appropriate. Finally, we choose to adopt the attribute solution because of its consistence rather than parsing the dataset name.

5.3.1 Numbering using counters

The counter programming model is one of the most used when it is necessary to count something. This because it is really linear and straightforward. Basically, it is only necessary to keep a counter variable and increase it by one any time that a new useful data for the context is found.

When writing the first dataset, i.e. when the group has just been created because it did not exist, it is not necessary to count the number of data sets. Obviously the name of the dataset will be *Dataset1* as we just created the group to write the first dataset in it.

However, when the group already exists, i.e. there are one or more datasets, we must count how many datasets have been already stored in the group. This is done using the `H5L` interface, in particular the function `H5Literate`. This function allows to visit in a recursive way the object passed as input parameter and execute a user-defined function

for each element found. These user function will update a counter variable initialized at zero. Inside the function, it is possible to determine the type of the object that is being visited and thus filter other groups from actual data sets. In case that the object is a dataset, the function will increase the counter by one. When the recursion is finished we obtained the number of datasets within the given group. At this point we can ensure a unique dataset identifier by just adding one to the counter obtained.

5.3.2 Numbering using attributes

The HDF5 attributes are intended to store metadata. Therefore they are really useful to store information to describe the primary data objects which is just the case.

As in the implementation described in 5.3.1, when writing the first dataset into a group that has just been created it is obvious that there were not previous data sets for this group and therefore the dataset just have the first *id*, i.e. `Dataset1`. However, in this case we must write the attribute `NUM_OF_DATASET_ATT` once the group is created. Once the attribute is created the value is written, obviously one for this case, and closed to free the HDF5 resources.

When writing a dataset into a preexisting group the attribute `NUM_OF_DATASET_ATT` is opened and read. This will provide the number of data sets already written into the given group. With this information the new data set is created, the attribute updated and all the resources closed.

5.4 Java Native Interface

In order to communicate with the *C* code it is necessary to define and develop a JNI interface. The JNI has been designed to allow Java applications to access libraries written in different languages which is our current scenario. In general, it is also used when it is not possible to implement some features using only Java. Finally, it can be also necessary to implement and to maintain some parts of the code that are time-critical, too low-level or architecture dependent to be implemented in Java.

However, native libraries must be compiled and linked in all the target platforms in order to work properly. This might lead to a certain loss of portability. Also, native libraries

must be loaded into the Virtual Machine (VM) with the method `System.loadLibrary()` before they can be used. Once the library has been loaded the Java VM can access the native functions implementation. These native functions are defined by using the keyword *native*.

For each native function declared in Java, which just defines a *prototype*, a native implementation is generated following a predefined syntax [17]:
`Java_package_and_classname_function_name(JNI arguments)`
The function receives the defined arguments plus two additional arguments that are always passed as input to the native methods:

- `JNIEnv`: This is the JNI interface pointer and is always the first argument passed to native methods.
- `jobject`: A reference to the object for non-static native methods. Otherwise, a reference to its Java class.

JNI allows the usage of all the primitive types, passed by value, and any Java object which are passed by reference. Native functions may return values back to the Java VM.

In our scenario the Java data objects must be stored in HDF5 files. To do so the DAL will use the JNI interface functions to access the C code that will access the HDF5 libraries to write the data into the actual file. The reverse procedure will be used to retrieve data from the HDF5 file.

To write the JNI code we used the following procedure. First we write the Java class that will access the C code. Inside this class, we load the native library by using the `System.LoadLibrary()` call. The parameter of this function is the name of the library which will be mapped into the corresponding *.so* file. The library should be included in the Java VM library path in order to be found. In our case, for simplicity, all the native methods are declared in a given Java class.

The next step is to compile the java class to create the corresponding *.class* file. Once this is done the *javah* utility is executed over the file. This creates the header file, so a *.h*, which contains the prototypes for all the native methods.

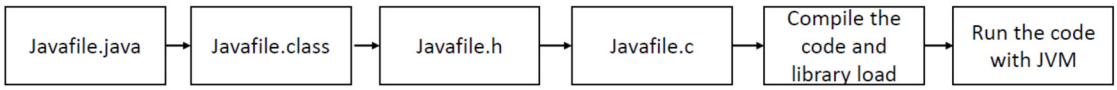

FIGURE 5.2: Communication scheme between the Java application with the HDF5 library through JNI

Once the header file is available, all the prototypes must be implemented, in this case using C. Java provides the relevant libraries to be included that provide access to the required types and functions.

Finally, the native code must be compiled to produce the desired shared libraries. In our case we use the HDF5 `gcc` wrapper, named `h5cc` to compile the code.

Fig. 5.2 shows a graphical representation of this process.

Additionally, taking advantage of the JNI features we implemented checks within the native functions that throw a Java exception in case of failure or error. This allows easier handling of errors within the Java code. All the code has been documented with the corresponding inline comments.

The JNI interface has been developed in coordination with the the Barcelona IDU/DPCB team, in order to integrate it with the current operational system.

5.5 DpcbTools interface

The Java part of this project has been integrated into a preexisting operational software library named *DpcbTools* and therefore it has been done in coordination with the Barcelona IDU/DPCB team. This has allowed to integrate the work done in this project into the existing DAL architecture in the most efficient way.

In particular, two main classes have been implemented in order to read and write the data: `FileHdf5Archiver` and `FileArchiveHdf5FileReader`.

The `FileHdf5Archiver` class is in charge of writing the data into the file. It is composed of all the internal variables and methods needed to handle the writing procedure. The constructor initializes class status variables including a flag to indicate which type of HDF5 file should be created, either *flat* or *hierarchical*. The value of the flag can always

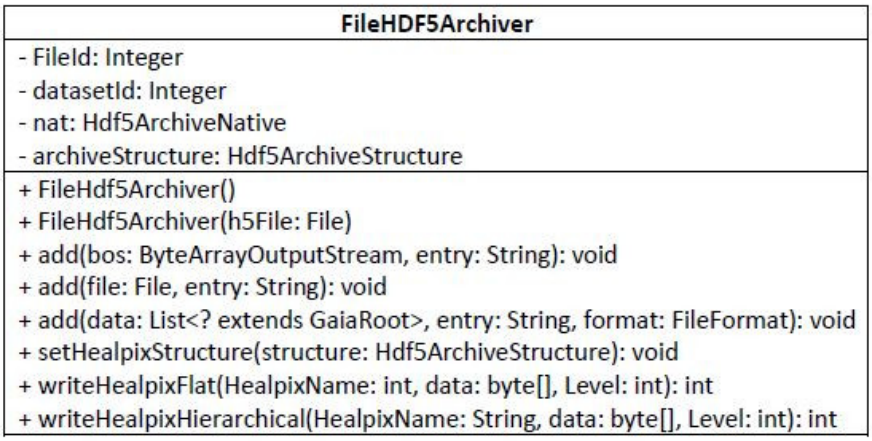

FIGURE 5.3: Simplified UML diagram for the FileHDF5Archiver class

be modified at runtime by using the corresponding `setHealpixStructure` function. When this class is created the HDF5 file is initialized and the corresponding identifier for the file is stored within the class. This identifier represents the HDF5 location.

To write data into the file, several `add` methods, which take different parameters according to different input data formats, have been implemented. Basically, these functions process the data and retrieve the HEALPix name, index and level. Once this is available it will call the correct native function (either the hierarchical or flat implementation) to write the data array into the file. Once the data has been written as a opaque dataset on the file, it is necessary to add the information regarding how many objects the dataset contains. This is required because, since the data is written as a opaque data set, it is not possible to know, before reading back the byte arrays, how many elements are stored in a given dataset. However, this information is fundamental for the right integration of the native code within the existing DAL. Thus, an attribute named `NUM_OF_ELEMENT_ATTR` is added to each dataset by calling the appropriate native function.

Fig. 5.3 shows a simplified UML diagram of the `FileHDF5Archiver` class, where the principal methods and attributes have been listed.

The `FileArchiveHdf5FileReader` class is responsible for the reading part of the code. The class constructor requires an HDF5 file and a HEALPix. When the constructor is called, the internal variables are initialized, the HEALPix name, index and level are computed and the file is opened.

Once all the values are set, the first step is to retrieve the list of datasets that might contain information for the desired HEALPix. However, this is done differently for the *flat* and *hierarchical* file structures. Therefore to call the correct native function the `FLAT_FLAG_ATT` attribute on the file is read. Then the `getNameListFlat` or the `getNameList` are called. Both of them will collect the list of full paths of all the datasets that match the given HEALPix.The initial retrieval of the list of datasets actually enables certain features that the DAL Application Programming Interface (API) defines, as the progressive loading of information. However, having the list of full paths for all the relevant data sets also allows to use a single native function to read the datasets from the HDF5 file.

The DpcbTools DAL defines several methods to access the data. Basically the user can request one object at a time or the full list of objects at once. In any case, first the number of Java objects that are going to be retrieved are counted calling the method `getCount`. This is done by calling the corresponding native method of the JNI interface and the information is stored within the reader class.

The user can easily request the stored objects one by one. Using the method `hasNext` it is possible to know if there is data left to be delivered to the user. The function basically checks if the data that has been delivered to the user is less then the data stored in the file. The `getNextObject` function will return the next object to the user. To do so it first checks if the internal object buffer within the class is empty. If it is empty the method will use the corresponding native function to read the next dataset from the list of datasets to be read. Once the byte array has been retrieved from the HDF5 file the function will deserialize the data, fill the object buffer and return the first object. This implementation allows to store in memory the minimal amount of objects. If the function is called and the buffer is not empty it will just return the next object in the buffer.

It is possible to retrieve the full list of objects at once by calling the method `getAllObjects`. The function will just iterate over the `hasNext` and `getNextObject` functions to fill an list with all the objects an return it to the user.

Fig. 5.4 shows a simplified UML diagram for the `FileArchiveHdf5FileReader` class. It shows the main methods and attributes relevant for this project.

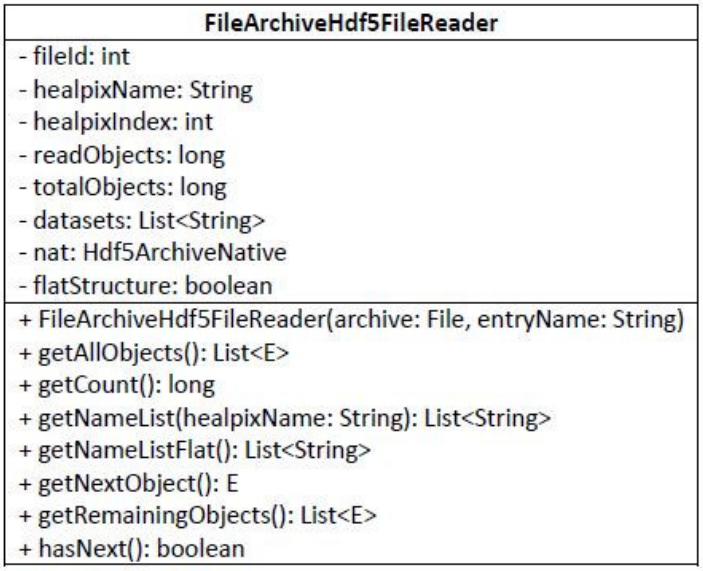

FIGURE 5.4: Simplified UML diagram for the FileArchiveHdf5FileReader class

Finally, a *Unit Test* class has been implemented in order to test the code with the automated build system available at DPCB. The `FileHdf5ArchiverTest` class tests the correct behavior of the code. In total, 16 tests have been implemented in order to check all the possible cases of writing and reading. In particular, for both the *flat* and *hierarchical* structure both reading and writing is tested. The tests ensure that the retrieved data is exactly the same as the written one. There are tests that work at different HEALPix levels.

Chapter 6

Tests and Results

This chapter details the tests and performance evaluation carried out for the implemented solutions. The focus has been specially on the evaluation of the I/O performance. First, as described in Section 6.1, we evaluate both the *hierarchical* and *flat* file structures. Section 6.2 describes the test scenarios and Section 6.3 shows the results for the designed tests.

6.1 File structure evaluation tests

The first test activity for this project was to test the feasibility and performance when creating the *complete* HDF5 file structures for both the *hierarchical* and *flat* designs. The tests were performed using a very simple C program which creates the HEALPix pixels at a given level and the corresponding groups. The program takes the HEALPix target level as a single argument. The maximum level tested is 12. The tests report the time required to create the file and the final HDF5 file size, only with the groups (no data or datasets created).

Figs. 6.1 and 6.2 show an example of the structures created. The main difference between them is the distribution of the groups trough the file. In the *flat* structure all the groups are at the same level, contained in the root `/`. In the *hierarchical* the groups are distributed according to the HEALPix hierarchy, descending until the required level. Further information regarding the HEALPix structures is available in Chapter 5.

```
HDF5 "fileHDF5.h5" {
  GROUP "/" {
    GROUP "E0" {
     GROUP "E00" {
     }
     ...
     GROUP "E03" {
     }
    }
    GROUP "E1" {
      ...
    }
    ...
    GROUP "S3" {
      ...
      GROUP "S33" {
      }
    }
  }
}
```

FIGURE 6.1: Hierarchical test structure example for Healpix level 1

```
HDF5 "fileHDF5.h5" {
  GROUP "/" {
    GROUP "0" {
    }
    GROUP "1" {
    }
    GROUP "10" {
    }
    ...
    GROUP "7" {
    }
    GROUP "8" {
    }
    GROUP "9" {
    }
  }
}
```

FIGURE 6.2: Flat test structure example for Healpix level 1

Note that these tests create all the HEALPix codes for a given level. In other words, we are evaluating the worst case scenario where there is data for every pixel at a given level and this is all stored in a single file. However, this will most likely never be the case in real operations for practical reasons. The data will most likely be split into several files for convenience.

Table 6.1 shows the results from the tests with the *hierarchical* file structure. The creation time and the file size grow up exponentially respect to the HEALPix level. The time and the file size needed for the creation of the file becomes significant when higher levels are tested.

Table 6.2 shows the results for the same performance tests for the *flat* file structure. The file creation time has a significant difference when comparing with Table 6.1. When the level is lower than 6, the difference between the creation time in both structures is minimum; in some cases even smaller for the *flat* structure. However, from level 6 onward, while the creation time for the *hierarchical* keeps growing at the same ratio, the *flat* structure creation time increases dramatically. Therefore, the tests for the *flat* structure from level 9 onward have been launched but were aborted after more than 24 hours of execution without having finished.

Most likely the creation time is longer in the *flat* case all the groups are created within the same one. In fact, although the number of groups created at the target level is the

Level	Creation time [s]	File Size	Number of groups	Number of Healpix pixels
0	0.012	10 KB	12	12
1	0.017	48 KB	60	48
2	0.013	199 KB	252	192
3	0.028	802 KB	1020	768
4	0.093	3 MB	4092	3072
5	0.378	13 MB	16380	12288
6	1.654	52 MB	65532	49152
7	16.563	206 MB	262140	196608
8	27.610	824 MB	1048572	786432
9	233.954	3 GB	4194300	3145728
10	461.757	13 GB	16777212	12582912
11	N/A	N/A	67108860	50331648
12	N/A	N/A	268435452	201326592

TABLE 6.1: Hierarchical file structure performance by Healpix level

Level	Creation time [s]	File Size	Number of groups	Number of Healpix pixels
0	0.018	10 KB	12	12
1	0.014	39 KB	48	48
2	0.020	155 KB	492	492
3	0.027	615 KB	768	768
4	0.075	2 MB	3072	3072
5	0.253	10 MB	12288	12288
6	1.177	39 MB	49152	49152
7	4.731	157 MB	196608	196608
8	22.163	629 MB	786432	786432
9	12125.837	3GB	3145728	3145728
10	N/A	N/A	12582912	12582912
11	N/A	N/A	50331648	50331648
12	N/A	N/A	201326592	201326592

TABLE 6.2: Flat file structure performance by Healpix level

same for both the solutions, their distribution within the file is different.

After the tests carried out the only reasonable motivation for this behavior is the computational cost for the HDF5 library to check the existence of a group before it is created. When groups are distributed in a hierarchical way the number of groups within any group is kept small – always less than 4. On the other hand, when the *flat* structure is created, all the groups are created inside the root. Each time that a new group is created the library has to check if it already exists inside this group. That becomes very inefficient when the number of groups grows.

6.2 Test cases

This section describes in detail the tests that have been implemented to evaluate the integration of our solutions into the DpcbTools framework. In order to evaluate the performance a specific Java task has been created, named `ArchiveFormatPerTask`. The class is designed to load a set of data into memory, write it in the different archive formats available, read the files back and ensure the contents are the expected ones.

The first step is the initialization of the execution environment. All variables, timers and statistics are initialized. Afterwards, the properties are loaded from a configuration file to setup the task accordingly. The configuration specifies parameters like the number of times each test should be performed (averaging the results) or the maximum number of files to be processed. By repeating the same tests and averaging the results we remove possible outliers due to specific conditions of the system.

The test performs the following operations:

1. Write archive file, according to the specific parameters (level, format, etc.)
2. Read archive sequentially, in the same order as it has been written.
3. Read archive requesting the pixels randomly.

These operations are performed for several different HEALPix levels and for each archive type available in DpcbTools (`HDF5`, `ZIP`, `TAR`). For the case of HDF5 we test both the *hierarchical* and *flat* solutions. The main difference between the formats is how the data is stored on the files. In fact, `ZIP` and `TAR` file formats are mainly used as a container to reduce the number of files.

To test the reading rate two different situations are tested. The first test is to measure the read rate when reading the whole archive file when the DAL is working at the specified target read level. The DAL then reads pixel by pixel at the target read level the information and returns it all together to the user. The timing is computed between the first read operation and the last read operation. This is therefore the time it takes to read all the information in the file when working at the specified level. On the other hand the second test measures the rate obtained when reading single pixels at the target read level. This measures the read rate when the user just wants a certain the

information for a certain HEALPix contained within the file. This is actually the most common situation in actual operations. In this case the timing is done between the start of the read operation and the moment just after the information (for that pixel) has been returned to the end user.

As the final execution of IDU is on MareNostrum the tests have been carried out in the same infrastructure in order to provide meaningful results. The MareNostrum nodes have two different file systems: a local 500 GB IBM hard drive, and a shared GPFS file system that can accessed by all the nodes. As both file systems have different response all the tests have been run against both. The tests that have been run again the local file system are named with the keyword *local* while the ones run against the GPFS file system are named with *global*.

All the operations in the test are timed and the information regarding the write and read rates is stored for each test. At the end of the test the accumulated information is processed and reported into the log file.

6.3 Results

This section reports the results obtained when running the tests specified in Section 6.2. For this tests we performed 3 iterations for each single tests and averaged the results to avoid possible momentary file system or computer issues.

The tests are run over two different types of data: `AstroObservation` and `Match` records. `AstroObservation` records have a mean size of 2.4kb, while `Match` records are much smaller with only 40b per record in average. The tests have been run with different data types in order to evaluate the differences in response time and performance. In this case, the size of both the sets of data is about 1GB.

The results have been summarized in tables that show the different combinations and their performance. In particular, local operations against the physical hard disk in the node are referred to as local. Tests against the shared file system (GPFS in MareNostrum) are referred as global. In each table is possible to compare the different formats that have been tested, where **HDF5_H** stands for HDF5 Hierarchical solution and **HDF5_F** refers to the HDF5 flat implementation.

Level	TAR	ZIP	HDF5_H	HDF5_F
0	22.05	16.98	21.77	**22.50**
2	21.84	17.04	21.92	**22.43**
4	22.28	17.06	21.98	**22.41**
6	**22.51**	17.21	22.43	16.37

TABLE 6.3: Write results on the GPFS file system in [MiB/s] for AstroObservation data. Best results for each level are marked in bold.

Level	TAR	ZIP	HDF5_H	HDF5_F
0	**22.78**	18.00	22.45	22.32
2	22.50	18.11	22.40	**22.54**
4	22.58	18.27	22.54	**22.60**
6	**23.19**	18.45	16.52	16.46

TABLE 6.4: Write results on the local file system in [MiB/s] for AstroObservation data. Best results for each level are marked in bold.

Tables 6.3, 6.4, 6.5 and 6.6 summarize the results for the write operation tests. The results show the write rate in MiB/s for both the `AstroObservation` and `Match` and the GPFS and the local file system. We show the results for all the available archive solutions.

As it can be seen in Tables 6.3, 6.4, 6.5 and 6.6 there are no a major performance differences between the different solutions. The results are of the same order of magnitude. However, we can appreciate a small performance drop when the ZIP format in used. Also, we can see how overall the best to performance is obtained when using the TAR solution. This is somehow expected as the TAR format only appends the data to the file without doing any extra operations. Therefore it is normal that the write rate is usually better. The HDF5 formats however, must write the data in the specified HEALPix structure and this has a computational cost when using higher HEALPix levels.

In general we can see that the write performance of when using the new HDF5 formats is comparable in most cases to use the TAR format, and almost always even better than using the ZIP implementation.

Therefore, as the difference between the TAR alternative and the two new solutions is almost null, so it is totally reasonable to adopt one of the new solutions in order to have a global considerable improvement in the I/O performances.

Tables 6.7, 6.8, 6.9 and 6.10 show the results for the reading speed tests when reading the whole file. In these tests the data is written and read at different levels but the

Level	TAR	ZIP	HDF5_H	HDF5_F
0	**23.71**	17.87	22.92	23.43
2	23.70	17.80	**23.71**	23.67
4	**23.59**	17.95	23.32	23.56
6	23.37	18.47	**24.40**	23.29

TABLE 6.5: Write results on the GPFS file system in [MiB/s] for Match data. Best results for each level are marked in bold.

Level	TAR	ZIP	HDF5_H	HDF5_F
0	23.45	19.00	**23.53**	23.39
2	23.84	19.00	**24.03**	23.92
4	**23.73**	19.03	23.65	23.44
6	**24.98**	19.47	23.19	23.32

TABLE 6.6: Write results on the local file system in [MiB/s] for Match data. Best results for each level are marked in bold.

whole file is read.

Note how the rate decreases when the HEALPix level for the write and read operation are different. In fact, the best performance is usually reached when the writing and reading are executed at the same level. It is interesting to see how the rate decreases each time the read operation is performed in a lower level respect to the write operation. Thus, if the user wants to retrieve all the data within a given file the best option is to store it at the highest level possible and read it at this same level. This is obviously a logical conclusion as in the end the number of operations is reduced. It is also remarkable that the read rate is in most of the cases better with the new HDF5 implementations.

Tables 6.11, 6.12, 6.13 and 6.14 show the performance results when requesting a single pixel at the given target read level when the file has been written at the specified write level. In this case only the time that is consumed to fetch the required HEALPix data is considered, as opposed to the total time required to read the file at a given HEALPix level that was reported in the previous tests. This simulates the situation where the user wants to access only part of the data stored within the file. In the cases where the data is stored at a given HEALPix level but the pixel requested has a higher level, so more detailed spatial resolution, the Java code must filter the data after it has been read. For example, when data has been stored at HEALPix level 0, say N0, but the user requests the HEALPix N0321, the Java code must read the entire N0 pixel and then filter the contents that have the N0321 pixel. This time is also considered part of the time taken to read the data and deliver it to the user.

Write Level	Read Level	TAR	ZIP	HDF5_H	HDF5_F
0	**0**	81.52	83.55	**147.96**	146.66
	2	24.57	26.44	**46.14**	45.56
	4	5.38	5.65	**10.05**	**10.05**
	6	0.96	1.00	**1.74**	**1.74**
2	**0**	68.42	82.87	**143.07**	141.16
	2	53.63	81.57	128.31	**137.07**
	4	12.38	17.96	**30.69**	30.51
	6	2.07	2.79	**4.83**	4.82
4	**0**	47.71	81.81	**132.27**	96.09
	2	39.33	80.74	**121.83**	95.84
	4	21.73	76.25	**77.59**	70.77
	6	3.92	10.68	**12.58**	11.38
6	**0**	15.75	**69.79**	25.12	8.48
	2	14.45	**67.12**	18.27	7.88
	4	11.42	**62.15**	8.00	5.15
	6	4.94	**49.17**	1.94	1.81

TABLE 6.7: Read results on the GPFS file system in [MiB/s] for AstroObservation data. Best results for each level are marked in bold.

Write Level	Read Level	TAR	ZIP	HDF5_H	HDF5_F
0	**0**	86.71	84.43	**149.31**	147.41
	2	26.49	26.72	**46.86**	46.63
	4	5.75	5.76	10.10	**10.12**
	6	1.02	1.01	1.75	**1.76**
2	**0**	75.36	83.91	**146.47**	146.34
	2	62.80	83.17	**141.18**	140.83
	4	14.34	18.35	**31.13**	30.89
	6	2.34	2.84	**4.93**	4.88
4	**0**	56.48	82.84	**138.85**	109.92
	2	48.97	82.23	**126.37**	104.63
	4	27.96	78.60	**91.53**	81.87
	6	5.07	11.01	**13.75**	12.58
6	**0**	21.09	**69.03**	26.03	10.35
	2	20.06	**70.25**	19.05	9.27
	4	15.42	**67.04**	8.51	5.87
	6	7.02	**52.39**	2.08	1.96

TABLE 6.8: Read results on the local file system in [MiB/s] for AstroObservation data. Best results for each level are marked in bold.

Write Level	Read Level	TAR	ZIP	HDF5_H	HDF5_F
0	**0**	47.78	50.40	51.66	**54.25**
	2	48.05	55.26	53.40	**63.86**
	4	7.75	8.03	8.73	**8.84**
	6	0.51	0.53	**0.57**	**0.57**
2	**0**	47.95	50.91	54.14	**56.15**
	2	48.40	50.99	**57.74**	53.05
	4	7.81	7.99	8.48	**8.79**
	6	0.52	0.53	0.58	**0.78**
4	**0**	47.12	44.78	61.55	**63.39**
	2	45.81	54.76	62.62	**65.43**
	4	34.92	44.05	60.18	**61.36**
	6	2.84	3.79	4.63	**4.65**
6	**0**	20.22	53.06	**59.23**	38.99
	2	20.58	49.50	**58.93**	38.52
	4	18.97	48.64	**54.55**	34.17
	6	8.26	**45.49**	27.09	21.07

TABLE 6.9: Read results on the GPFS file system in [MiB/s] for Match data. Best results for each level are marked in bold.

Write Level	Read Level	TAR	ZIP	HDF5_H	HDF5_F
0	**0**	52.23	51.67	**61.69**	53.87
	2	51.96	56.62	55.75	**59.19**
	4	8.17	8.17	**8.82**	8.78
	6	0.54	0.54	**0.57**	**0.57**
2	**0**	51.10	54.02	55.01	**63.59**
	2	50.55	53.78	**55.45**	54.32
	4	8.01	8.23	**8.87**	8.78
	6	0.53	0.55	**0.58**	0.57
4	**0**	50.51	53.60	65.82	**66.66**
	2	47.66	53.09	**66.27**	66.23
	4	41.17	41.16	51.38	**67.33**
	6	3.11	3.77	**4.76**	4.73
6	**0**	25.37	**55.67**	54.70	42.38
	2	25.53	52.94	**55.06**	41.87
	4	23.35	**51.36**	44.08	36.03
	6	10.23	**37.72**	25.91	23.46

TABLE 6.10: Read results on the local file system in [MiB/s] for Match data. Best results for each level are marked in bold.

Write Level	Read Level	TAR	ZIP	HDF5_H	HDF5_F
0	**0**	68.94	81.23	136.70	**142.72**
	2	17.63	20.66	**36.50**	36.34
	4	3.82	4.40	7.47	**7.76**
	6	0.71	0.83	**1.44**	**1.44**
2	**0**	55.62	80.53	**136.90**	133.95
	2	37.35	71.50	108.01	**108.08**
	4	7.98	15.19	**24.19**	23.84
	6	1.50	2.88	**4.47**	4.39
4	**0**	37.81	76.37	**117.88**	78.05
	2	27.25	68.52	**82.63**	60.93
	4	14.82	53.84	**74.09**	42.43
	6	2.84	**10.09**	8.84	7.91
6	**0**	15.80	**58.45**	17.35	7.45
	2	12.76	**51.26**	12.09	5.67
	4	8.42	**41.39**	6.22	3.85
	6	4.28	**29.80**	1.95	1.82

TABLE 6.11: Read results on the GPFS file system in [MiB/s] for AstroObservation data for single read. Best results for each level are marked in bold.

In general, the HDF5-based alternatives performance is much better compared with the ZIP and TAR solutions. Specifically, when the data is read at a higher level than the level it has been written into the file, the performance for the HDF5 alternatives is significantly better than all other alternatives. For example, it is preferable to store the data at HEALPix level 4 if it must be read at some point at level 4 (even if sometimes it will be read at level 2) than storing it at level 2 and then trying to read it at level 4. Therefore it is always preferable to store the data with fine spatial resolution (high HEALPix level). This will allow the data to be read at the same or a lower level (less spatial resolution) with reasonable performances.

It is important to note the difference in the data rate when storing `AstroObservation` or `Match` data. For `AstroObservation` data, the rate obtained is higher than when storing smaller `Match` records. At the same time, the rate decreases faster from one level to the next for the `AstroObservation` data. Most likely this is a result of the different record size. A high record size might produce larger serialized arrays which might have a different behavior. This shall be explored in future developments of these solutions but is out of the scope of the current project.

Finally we also note how the ZIP file format performs substantially better when writing `AstroObservation` data at level 6, specially when requesting a single pixel. However, the data rate when reading `Match` data is not strongly affected. The reason for this

Write Level	Read Level	TAR	ZIP	HDF5_H	HDF5_F
0	**0**	75.57	82.96	144.95	**147.05**
	2	19.33	20.84	**36.95**	36.91
	4	4.12	4.46	**7.83**	7.80
	6	0.76	0.85	**1.45**	**1.45**
2	**0**	60.75	82.05	**141.39**	137.38
	2	44.63	73.95	**115.86**	112.16
	4	9.47	15.81	**25.35**	24.69
	6	1.75	0.22	**4.63**	4.49
4	**0**	44.01	79.00	**123.73**	87.75
	2	32.71	70.38	**91.14**	68.62
	4	18.31	56.09	**75.88**	48.33
	6	3.52	**10.42**	9.81	8.79
6	**0**	19.58	**62.49**	19.36	8.87
	2	16.26	**51.26**	13.33	6.71
	4	10.88	**43.94**	6.72	4.49
	6	5.70	**31.21**	2.09	1.97

TABLE 6.12: Read results on the local file system in [MiB/s] for AstroObservation data for single read. Best results for each level are marked in bold.

Write Level	Read Level	TAR	ZIP	HDF5_H	HDF5_F
0	**0**	53.50	59.49	75.30	**76.33**
	2	54.80	58.38	75.38	**77.64**
	4	11.87	13.36	**16.57**	16.27
	6	0.80	0.86	1.04	**1.43**
2	**0**	53.71	58.61	75.24	**77.12**
	2	53.69	59.42	75.84	**75.93**
	4	11.72	13.33	**16.12**	16.09
	6	0.79	0.86	**1.04**	**1.04**
4	**0**	51.07	58.13	**77.17**	76.76
	2	51.57	58.03	72.76	**74.34**
	4	45.33	59.09	47.33	**75.88**
	6	2.90	3.91	1.09	**5.03**
6	**0**	25.82	57.84	**64.77**	47.09
	2	26.06	58.08	**61.01**	45.50
	4	21.08	57.81	**57.92**	39.69
	6	9.57	**54.50**	29.92	25.66

TABLE 6.13: Read results on the GPFS file system in [MiB/s] for Match data for single read. Best results for each level are marked in bold.

Write Level	Read Level	TAR	ZIP	HDF5_H	HDF5_F
0	**0**	55.10	58.73	76.18	**76.38**
	2	52.01	59.22	76.50	**77.70**
	4	12.56	13.48	**16.46**	16.42
	6	0.82	0.87	**1.05**	1.04
	0	51.78	59.55	77.31	**77.36**
2	**2**	55.90	59.45	77.08	**77.15**
	4	12.69	12.91	**16.29**	16.17
	6	0.82	0.87	**1.05**	1.04
4	**0**	53.15	59.72	76.14	**76.16**
	2	54.38	59.76	**74.57**	74.37
	4	49.56	61.03	53.67	**76.54**
	6	3.19	3.96	**5.08**	5.06
6	**0**	30.27	58.49	**67.05**	51.23
	2	30.49	60.04	**64.49**	50.12
	4	25.41	59.49	**59.52**	43.63
	6	12.58	**56.15**	32.55	29.04

TABLE 6.14: Read results on the local file system in [MiB/s] for Match data for single read. Best results for each level are marked in bold.

is that the ZIP archive implementation uncompresses the whole ZIP file into a local temporary directory until the archive file is closed. Thus, subsequent access to the data is faster. This obviously consumes local space in the local node disk. How this might affect performance when used in real IDU operations must be investigated in the future.

Chapter 7

Conclusions and future work

7.1 Conclusions

The scope of the Gaia mission will lead to the acquisition of an enormous quantity of data in order to produce the final catalogue. The amount of information that will be retrieved, for more than 10^{11} detections, requires specific and efficient DAL solutions.

The previous DAL implementation available within the software at DPCB was based on a custom file format, namely GBIN. Two archive solutions were also available which used TAR or ZIP formats. Neither the GBIN or any of the previous archive solutions were very efficient when storing or accessing the data according to a spatial distribution.

The need to have a proper organization of the data, based on their spatial distribution, led to the use of the HEALPix library to optimize the spatial access. We also considered the challenges that arise from using the MareNostrum III supercomputer at BSC to execute IDU. The choice has been to use the HDF5 library, accessing it through the JNI.

Two solutions have been implemented, using HDF5, that optimize the DAL, specially for spatially distributed data. The difference between both solutions is how the file is structured. The first solution follows the HEALPix hierarchy implementing it within the HDF5 files. The second solution stores the data in a flat structure within the HDF5 file in groups that have a unique HEALPix index. Both solutions have been successfully

integrated into the operational code at DPCB and might be used in future executions of IDU.

The performance has been evaluated with several tests that reported the read and write rates in two different file systems. The results have been compared with the previous solutions available within DPCB. This allowed to see the improvements and limitations delivered by the new alternatives.

The write operations do not report major differences between the different implementations. All the implementations report similar write rates except for the ZIP which has the lowest performance.

The test results report a substantial increase on the read rate with respect to the previous ZIP and TAR implementations. The TAR implementation delivers, in general, the worst performance figures. The difference between both HDF5 implementations is rather small. However, we concluded that overall the hierarchical one is preferable.

When comparing the performance of random data access within the archive files the tests report significantly better behavior of the proposed HDF5 solutions than the previous TAR and ZIP alternatives. In many cases the performance is increased by more than 50% when compared with the previous solutions. This is very significant as the data is accessed while processing IDU tasks, which usually request parts of the data as required, instead of entire files.

Finally, the work presented has largely contributed to the optimization of the DAL and it will help to collect and store the data in a much more efficient way. The data can now be stored more efficiently according to their spatial distribution in fewer well structured files.

7.2 Future work

This project has been a first exploration of the use of HDF5 as a solution to store the spatial data efficiently that should pave the way to further improvements and new developments at DPCB.

In the near future DPCB will test more exhaustively the new HDF5 implementations. First by using them in the next testing campaigns and comparing the results with the

previous executions. This will lead to more accurate performance assessments of the level of improvement that might be delivered by the new solutions. However, executing IDU tasks with operational volumes has a significant cost both in terms of computing time (several hundred thousands of hours) and also human resources that must be devoted to it. For this reason these tests will be included in the next major IDU testing campaign scheduled by the end of the year.

There are two aspects which have not been possible to explore in this project and that must be investigated. First, the difference in performance when storing data types with different record sizes. The tests carried out in this project show differences that must be well understood and might require some calibrations of the dataset size, buffer sizes, etc. in the future. Another result that must be investigated is the performance of the ZIP implementation in some of the tested situations.

Yet another improvement that would be very beneficial for the usage of the data at DPCB is to include time information within the spatially distributed data structure. In the proposed designs the HDF5 datasets were just numbered sequentially. However, for data that has both spatial and time information, such as `AstroObservation` records, the datasets could contain time information as well. This would solve the very inconvenient situation that arises when there are some tasks which require access to the data according to its HEALPix index but other require access according to time ranges. Currently if the information has been written according to the HEALPix structure (with any of the formats) but the user needs to access some specific time range it is forced to read all the HEALPix and filter the data. If this time information would be included into the file structure the user would just be required to open the file and check if there is information for the relevant time range, and therefore would be able to avoid reading and filtering all the data.

Finally, an aspect that can be explored in the future is the native compression algorithms provided by HDF5. In the current implementation the serialized Java data objects are compressed with ZIP before being sent to the JNI to be written in the HDF5 file. However, it would be interesting to analyze how the different compression algorithms that can be integrated as HDF5 filters perform.

Bibliography

[1] Learning from 25 years of the extensible n-dimensional data format. 2014.

[2] E. Bray P. Greenfield, M. Droettboom. Asdf: A new data format for astronomy. *Astronomy and Computing*, 2015. `http://www.digitalpreservation.gov/formats/fdd/fdd000317.shtml`.

[3] Fits - flexible image transport system. `http://www.digitalpreservation.gov/formats/fdd/fdd000317.shtml`, 2013.

[4] Fits - theastronomical image and table format. `http://fits.gsfc.nasa.gov/fits_documentation.html`, 2015.

[5] The gaia mission (gaia page at eoportal website). `https://directory.eoportal.org/web/eoportal/satellite-missions/g/gaia`, 2016.

[6] Gaia page at ESA science & technology website. `http://sci.esa.int/gaia/`, 2016.

[7] Gaia - mapping the galaxy and watching our backyard. `http://www.esa.int/ESA_in_your_country/Ireland/Mapping_the_Galaxy_and_watching_our_backyard`, 2016.

[8] Javier Castañeda. High performance computing of massive astrometry and photometry data from gaia. *Programa de doctorat en Física - Línia de Recerca en Astronomia i Astrofísica*, sep 2015.

[9] Krzysztof M. Górski. Mapping the CMB sky: The BOOMERANG experiment. *arXiv:atro-ph*, 1999.

[10] C. L. Bennett. The microwave anisotropy probe mission. *The Astrophysical Journal*, 2002.

[11] Krzysztof M. Gorski. HEALPix: a framework for high-resolution discretization and fast analysis of data distributed on the sphere. *The Astrophysical Journal*, 622(2): 759, 2005.

[12] HEALPix - data analysis, simulations and visualization on the sphere. `http://healpix.sourceforge.net/`, 2016.

[13] Krzysztof M. Górski. The HEALPix primer. `http://healpix.sourceforge.net/pdf/intro.pdf`, 2015.

[14] Pierre Fernique. MOC-HEALPix multi-order coverage map version 1.0. *arXiv preprint arXiv:1505.02937*, 2015.

[15] The HDF Group. Hierarchical data format, version 5. `http://www.hdfgroup.org/HDF5/`, 1997-2016.

[16] HDF Group et al. HDF5 user's guide. `https://www.hdfgroup.org/HDF5/doc/PSandPDF/HDF5_UsersGuide.PDF`, 2011.

[17] JNI - java native interface. `http://docs.oracle.com/javase/7/docs/technotes/guides/jni/spec/jniTOC.html`, 2016.