TRABAJO DE FIN DE MÁSTER
MÁSTER EN COMPUTACIÓN PARALELA Y DISTRIBUIDA

ANÁLISIS DEL RENDIMIENTO Y OPTIMIZACIÓN DE UN CÓDIGO NUMÉRICO PARALELO PARA EL CÁLCULO DE LA EMISIÓN EN PLASMAS ASTROFÍSICOS

Autor
Carmen Aloy Torás

Directores
Dr. Petar Mimica
Dr. Vicent Vidal Gimeno

Septiembre de 2013

Agradecimientos

Quisiera agradecer agradecer al Profesor Miguel A. Aloy, miembro del Grupo de Astrofísica Relativista (GAR) , la idea inicial del presente trabajo, los valiosos consejos y sugerencias dadas durante la realización del mismo, así como la posibilidad de utilizar el clúster *Lluis Vives* a través de la cola del Grupo para la realización de pruebas.

También, al Dr. Petar Mimica, Co-Director de este trabajo, quisiera agradecerle todo el trabajo realizado en colaboración con el desarrollo del mismo. Dar las gracias por las ideas, sugerencias y consejos, por disponer de sus codigos fuentes sobre los que hacer modificaciones, por los ficheros de entrada necesarios para pruebas, por las correcciones y validaciones.

Y por último, agradecer al Profesor Vicent Vidal, por permitir el desarrollo de este trabajo en el Grupo de investigación en el que trabajo y co-tutorizarlo con el Dr. Petar Mimica.

Resumen

La presente memoria expone el trabajo de análisis del rendimiento y optimización de códigos numéricos paralelos, que realizan el cálculo de la emisión en plasmas astrofísicos, estudiados en el trabajo de investigación desarrollado por los miembros del Grupo de Astrofísica Relativista de la Universidad de Valencia, en el que actualmente colaboro como técnico de soporte a la investigación a través del Programa Nacional de Técnicos de Apoyo 2011 del Ministerio de Economía y Competitividad.

El trabajo de optimización se ha llevado a cabo tras un intenso estudio de las necesidades de mejora en ciertos puntos claves de la ejecución de estos códigos, sobretodo nos hemos centrado en la velocidad de ejecución, el consumo de memoria, el espacio de almacenamiento y los requerimientos de ancho de banda. Para ello hemos realizado un perfilado y un muestreado estos códigos en diferentes escenarios, todos ellos tomados de casos reales, y con los resultados obtenidos de este perfil y muestreo, más numerosos tests de ejecuciones con diferentes combinaciones de parámetros de entrada, hemos podido tomar decisiones sobre en qué puntos implementar las mejoras en los códigos y las opciones o configuraciones de entrada que dan lugar a resultados, optimizando los aspectos de velocidad, consumo, espacio, etc., antes mencionados.

Las mejoras implementadas serán explicadas detalladamente a lo largo de la presente memoria, mejoras tales como la optimización del tiempo total de ejecución y del tamaño de los ficheros de salida Cap. 2, en la paralelización de tareas que pueden simultanearse Cap. 3, optimización del uso de memoria Cap. 4 y optimización de la escritura paralela de los ficheros de salida Cap. 5.

Con todas estas optimizaciones, se pretende conseguir unos códigos numéricos altamente eficientes que puedan correr en supercomputadoras de la Red Española de Supercomputación consumiendo el mínimo de recursos posible de estas máquinas y que puedan proporcionar resultados a los investigadores en el menor tiempo posible para que puedan realizar su labor de análisis de los mismos.

Contents

Introduction

The work presented in this thesis focuses on the optimization of numerical tools which are essential for the research conducted by members of the Relativistic Astrophysics Group (RAG) at the University of Valencia. The research at RAG deals with the study of the astrophysical plasmas encountered in a wide variety of scenarios [1] .

Computations performed by the Relativistic Astrophysics Group

On the one hand the RAG studies the theoretical models of relativistic jets (in active galactic nuclei, or in progenitors of gamma-ray bursts) and compact stellar objects (neutron stars and black holes). On the other hand the electromagnetic and gravitational radiation emitted from such sources is analyzed, in order to be able to compare the theoretical models to observations.

In order to achieve these objectives the group strongly relies on the analysis and simulations of the most relevant physical processes, i.e. they shall accurately model the astrophysical scenarios at hand including radiative transfer in multidymensional numerical simulations. The constitutive areas of research of the group members and their broad objectives are the following:

- **Extragalactic jets** are among the most fascinating objects in the universe. They emerge from the centers of active galactic nuclei (AGN) and propagate with velocities very close to the speed of the light up to distances of thousands (in some cases even millions) of light years from their host galaxy. In the RAG their formation, acceleration and collimation mechanisms are studied, as well as their stability properties and their structure. This is achieved using one, two- and three-dimensional magneto-hydrodynamical simulations coupled to radiative transfer codes. To test the validity of the simulations observational

[1] www.uv.es/astrorela

data from the whole electromagnetic spectrum is used, with a special emphasis on the radio, optical, X-ray and γ-ray frequencies.

- **Gamma-ray bursts** are brief flashes of γ-rays which are, for a few seconds, the brightest objects in the sky when observed using γ-ray telescopes. Most probably the emission comes from a relativistic jet formed either during the collapse of a massive star or during the merger of two compact objects (two neutron stars or a neutron star and a black hole). We analyze the mechanism of jet formation and propagation using multidimensional magneto-hydrodynamical simulations, paying particular attention to the emission properties.

- **Compact stars** are of interest both as a sources of highly variable emission and as possible progenitors of gamma-ray bursts. We study the physics of neutrino production in dense superfluid matter, as well as neutrino transport in proto-neutron stars and in the accretion disks resulting from the merger of compact binaries. Furthermore, we investigate the thermal emission from magnetized neutron stars and their thermal and magnetic evolution. The results of our models are compared with observations in order to constrain the free parameters of the models.

- **Gravitational radiation** is an additional observational window into our universe (besides the electromagnetic spectrum). For a successful detection of gravitational radiation good theoretical predictions are indispensable. Using perturbative and full numerical relativity codes we compute the emission of gravitational radiation due to accretion processes on to compact objects, gravitational stellar core collapse to neutron stars and black holes, and pulsating and rapidly-rotating relativistic stars with the aim of helping the interpretation of future detections by Earth- and space-based gravitational wave detectors.

Use of computational resources

Members of the RAG commonly use the Spanish Supercomputing Network resources for their calculations. Currently, they are granted with a significant number of computing hours. In general, according to PRACE (Partnership for Advanced Computing in Europe) statistics, astrophysics occupies a 26 % of computational resources on some of the most powerful supercomputers in Europe [17], as is shown in Fig. 1

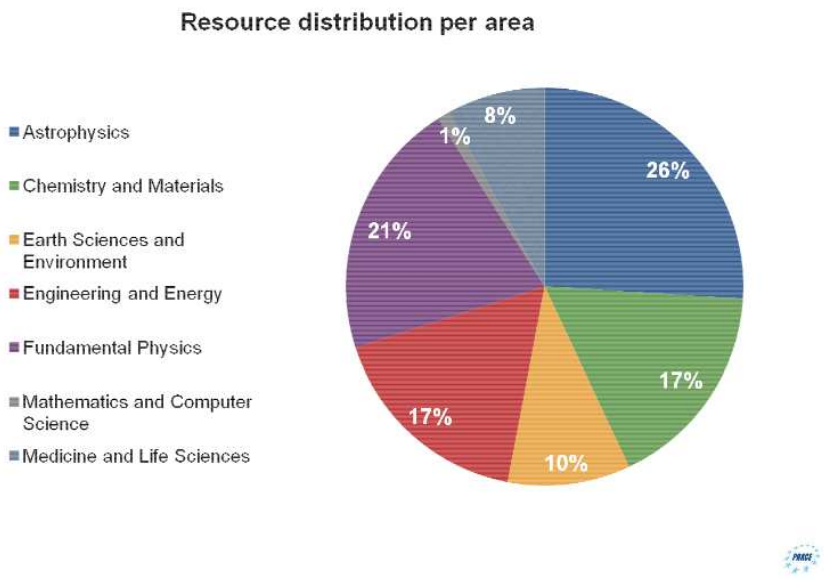

Figure 1: Distribution of computational resources per research area. Source: BSC-RES-PRACE

Optimization of the use of computational resources

Due to the complexity of the physics the demand of computational resources needed to solve the problems of interest is very high. Therefore, a high efficiency of astrophysical codes is in high demand. Common ways to achieve high efficiency are:

- **Code optimization**: by reviewing the source code we can determine if it can be improved in terms of clock cycles required to execute, subsequently rewriting parts of all of the source code as needed.
- **Parallelization**: distribution of the work load among multiple simultaneously executing threads.
- **Memory usage optimization**: analysis of memory access operations and, if necessary, change of the data structures used in the code.
- **Input/output optimization**: reduction of the number and volume of reads and writes to the external storage is achieved by using compression, parallel/distributed file systems and the change of the data structures used.

From the above it follows that of the RAG requires an intensive use of the computational resources which must be optimized. Some additional com-

pelling reasons for the optimization are:

1. Reducing economic costs of using supercomputers.
2. Reducing environmental impact, that is a result of energy consumption of machines and facilities.
3. Accessing large European supercomputers, with the purpose to carry out very detailled simulations with a large number of processors and memory.

Computational efficiency

The result of the optimization is an increase in the computational efficiency. First, we need to define what efficiency is, so that it can be increased. For decades the notion of efficiency has been associated mostly with the computing speed and measured in FLOPS (floating point operations per second). However the faster our supercomputers, the more energy they consume, dissipating excessive amounts of heat. This in turn means they require additional cooling infrastructure, which also consumes electricity. For that reason the proper definition of the efficiency should not only contain the computational efficiency, but also the energy efficiency. A good approach is to consider the electrical power consumed while performing a given number of operations per second. The unit in which this efficiency is measured is Watts per FLOP. We have devoted a significant effort to optimize the energy consumption to reduce not only economic costs, but also the environmental costs, i.e. we want the usage of these machines to be as respectful as possible to the environment and contribute to climate change as little as possible.

Evolution of computational efficiency in Astronomy

Astronomy and astrophysics have made huge progresses in the centuries since the first telescope was assembled but, at least in spirit, the observations made by the most sensitives satellites today are not very different from the very rudimentary observations made by Galileo 400 years ago. In both cases one is asking a question, formulating an hypothesis, testing it against the observational data and, finally, validating or refuting it. This is, of course, nothing more than the application of the scientific method in Astronomy, but it also points out all of the difficulties of its implementation: Astronomy is an *observational* science and not an *experimental* one. Once a hypothesis is constructed to explain an observed phenomenology, one finds that it is rarely the only plausible one. In contrast with an experimental science, in

Astronomy it is not possible to set-up an experiment in order to validate or refute the hypothesis. Rather, additional observations in the future are needed to discriminate among many likely explanations.

Obviously, the extreme physical conditions characterizing compact astrophysical objects are impossible to reproduce in terrestrial laboratories under controlled conditions. Astronomical observations over many wavelengths reveal in fact that in highly curved spaces around black holes (BHs) or neutron stars (NSs), plasmas endowed with ultra-intense magnetic fields move at speeds close to that of light. In reality, the conditions are so extreme and the relativistic effects are so dominant that our common experience and intuitions are of no help and even our imagination goes astray. Yet, these conditions can be reproduced in a *computational-astrophysics laboratory*. This is not a laboratory in a standard sense but, rather, it is a *virtual* one, a laboratory where the instruments are complex nonlinear equations and numerical algorithms, the pieces of equipment are fast and parallel supercomputers, the experiments are simulations investigating vast spaces of parameters, and the observations are the results of the visualization of enormous datasets. For a recent review of the state of the art of the hydrodynamic simulations of relativistic jets see [4].

Modern Astrophysics is critically dependent on the numerical modeling of nature. The more sophisticated the modeling the more computational resources it needs. In practice, almost any field of Astrophysics requieres a huge number of computing hours on the largest supercomputers. Indeed, Astrophysics is the scientific area which has employed the largest chunk of computing hours in PRACE supercomputers ($\approx$ 26% of all the available resources; Source: presentation of David Vicente in Frontiers on Grid and Supercomputing. October 2012 [17]). Gaining access to PRACE computing [15] time means to apply for it and passing rigorous evaluation by the selection committees, not only assessing the scientific level of the proposal, but also the technical feasibility of the codes which are going to be used for the problem which they are attempting to solve. Therefore, the long term goal of the optimization in this project is to create stable, well tested and well optimized codes so that the researchers can obtain access to supercomputers more easily and use the limited computing time as efficiently as possible.

The aim of the present work

There are many astrophysical objects and scenarios whose modeling can be tackled by suitable fluid dynamics simulations. For instance, stars, the winds ejected by stars, the interstellar medium, the galaxies, the clusters of galaxies

and even the Universe as a whole, can be treated as fluids. Thus, hydrodynamic simulations are very common in Astrophysics. Furthermore, as mentioned above, some astrophysical objects and scenarios bring Physics to the extreme limits properly described by the General or Special theory of Relativity. To complicate even more the picture, some objects (e.g., relativistic jets, magnetars, etc.) are magnetized to such a degree that the magnetic field evolution dictates virtually all of its dynamics. When both dynamically relevant magnetic fields and a suitable relativistic fluid approximation can be used to model the object of interest, the governing equations of the system are those of the Relativistic Magneto-Hydrodynamics (RMHD).

It is quite common in Astrophysics that the observational signature of a system governed by the RMHD equations is of *non-thermal* nature. This means that the particles that emit the electromagnetic radiation, which we ultimately observe with our telescopes on Earth, are not in equilibrium with the *thermal* plasma[2] that provides the dynamics and hosts most of the inertia of the system (this is especially the case when we deal with relativistic sources, e.g., jets flowing at speeds very close to the speed of light). In a thermally emitting system the local temperature and the fluid density fully determine its emissivity and, therefore, the fluid dynamics of the (thermal) plasma fixes what we can observe. In contrast, in the non-thermal sources we require additional physics modeling and a very specific treatment to estimate its observational signature. Typically, one needs to add to the thermal plasma, whose dynamics is computed employing the RMHD equations, a separate population of non-thermal (e.g., electrons) particles which are linked to the thermal plasma by the action of the magnetic field. [3]

MRGENESIS [3, 10] is a numerical code that can model Astrophysical sources governed by the RMHD equations. In the language of the previous paragraphs, this code is used to perform numerical simulations of the systems of interest considering *only* the evolution of the thermal plasma. As commented above, most relativistic sources are non-thermal emitters and, hence, in order to compute the observational signature one needs to perform additional (numerical) modeling on top of the dynamics of the thermal plasma. With that purpose in mind we built the *SPEV* code [5]. *SPEV* is a pioneer numerical tool, which can compute the spectral evolution of non-thermal particles as they suffer synchrotron loses and/or adiabatic transformations

[2]Plasma is one of the four fundamental states of matter (the others being solid, liquid, and gas). Heating a gas may ionize its molecules or atoms (reducing or increasing the number of electrons in them), thus turning it into a plasma, which contains charged particles: positive ions and negative electrons or ions. Some common plasmas are found in stars and neon signs. In the universe, plasma is the most common state of matter for ordinary matter, most of which is in the rarefied intergalactic plasma (particularly intracluster medium) and in stars.

[3]Charged particles such as electrons and protons are restricted to move gyrating around magnetic field lines. Since all accelerated charges radiate, it is due to this gyration that they emit the observed electromagnetic radiation.

because of the change in the underlying plasma properties, and simultaneously, it can produce synthetic light curves and/or images of the considered problem (Fig. 2). In other words, *SPEV* can produce *synthetic* observations out of RMHD numerical models, that can be compared directly with actual Astrophysical observations.

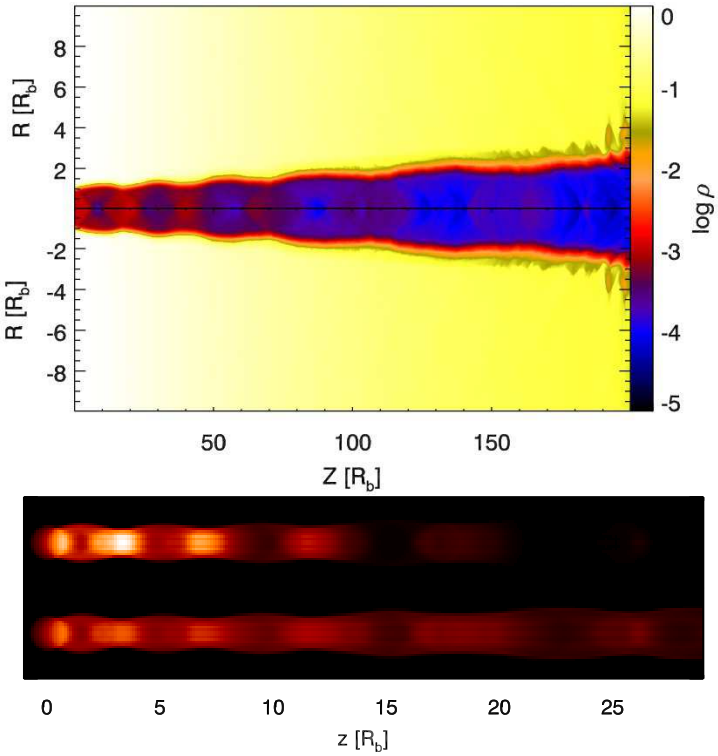

Figure 2: Top: density distribution (on a logarithmic scale) of an axisymmetric jet (red-to-blue colors) propagating into a stratified atmosphere (white-to-yellow), as simulated by *MRGENESIS* using a numerical grid with a resolution of 1800x80 zones. The jet is initially overpressured with respect to the atmosphere and develops a series of conical recollimation shocks (seen as an increase in density) in its channel. The length on both axes is in the units of the initial jet radius. Bottom: radio emission computed by *SPEV* using the data from the top pannel as an input. The image resolution is 270x18. The jet is assumed to be oriented so that its axis and the line of sight towards the art make a 10° angle. In the top image the radiation and adiabatic effects are taken into account, while in the bottom image only the adiabatic effects are considered. The recollimation shocks are identified with the bright knots of increased radio emission. The calculations have been performed in [5], and a special version of the figures produced for this thesis. We note that, although the resolution of the image is low by the standards of today's cameras, it is neverheless much higher than what can be obtained even by our largest radio-telescopes. The reason is that the radio jets are located at very large distances and are barely resolved using our most advanced instruments.

In many cases of interest, the radiative loses that the modeled system experiences due to the emission of non-thermal particles are negligible. Thus,

the underlying thermal plasma evolves adiabatically. From the technical point of view, this means that one can perform the calculation in two stages: first the simulation of the dynamical models (performed with *MRGENESIS*), and then the evaluation of the observational signature of the system (computed with *SPEV* in a *postprocessing* phase). In this thesis, we focus exclusively on this mode of work, i.e., the computation of numerical models governed by the RMHD equations is assumed to be *independent* of the evaluation of the non-thermal signature of such models. This allows us to perform independent optimizations of both numerical tools (*MRGENESIS* and *SPEV*). Indeed, the kind of optimizations needed in each of the two pieces of the work are quite different, as we will see in Chapters 3 and 4.

A typical problem in any fluid dynamical numerical simulation (and this includes RMHD simulations) is to discretize the space in a set of numerical cells or zones where one samples the *true* (i.e., continuous) solution. For instance, if our domain of interest is one-dimensional with size L, and extends from $x = 0$ to $x = L$, one would take N points, e.g., uniformly distributed, $x_i = (i + 1/2) \times h$ ($h \equiv L/N$; $i = 1, \ldots, N - 1$) where we desire to know the *approximate* evolution of the flow. Obviously, we cannot employ an arbitrarily large number of numerical cells, but suitable numerical methods (so called *convergent*) can be used in such a way that under mesh refinement (i.e., decreasing the grid size h or, equivalently, increasing the number of cells N) they converge to the physically acceptable (i.e, the *true*) solution of the system of equations. Naturally, a larger degree of realism is obtained when we sample the space with a larger number of numerical cells, with the drawback that the simulations become more computationally demanding. Therefore, we need to find a trade-off between the best resolution we could use and the available computing time. In systems with more than one spatial dimension, the problem of using the *minimally acceptable* numerical resolution exacerbates, since the number of computational zones grows as N^2 or N^3 depending on whether we are modeling two- or three-dimensional systems. There are several ways to ameliorate the problem of resolution in multidimensional fluid dynamics. One possibility is using non-uniform grids (sometimes *unstructured*) adapted to the dynamics we want to solve for. Another possibility is to use Adaptive Mesh Refinement (AMR), where the local properties of the fluid are examined to determine where higher resolution is needed. If that is the case the numerical mesh is dynamically (i.e., on run time) refined in the region that needs higher resolution. Obviously, one can use a mixture of both approaches, but then the numerical methods tend to become extremely complicated.

In the present work we will show the first steps towards implementing an existing AMR algorithm in *MRGENESIS*. Developing our own AMR-RMHD code would have taken a lot of time (estimated two to four years). Further-

more, the parallelization and optimization of AMR codes is extremely complex, since the difficulty is to reduce to the minimum the overhead brought by the management of the hierarchy of grids typical of AMR algorithms. Therefore, our choice has been to implement an existing AMR package adapting our code to the technical requirements of such package. In recent years, a number of public domain codes have appeared to provide the community with a basic infrastructure that allows building their own applications, which can make efficient use of massively parallel supercomputers. CHOMBO[4], is a prototype of such frameworks, maintained by the Applied Numerical Algorithms Group of the Lawrence Berkeley National Laboratory. The Multi-Scale Fluid-Kinetic Simulation Suite (MS-FLUKSS) is an example of the new codes that have been developed building upon the CHOMBO framework, which ensures efficient parallelization and dynamic load balancing on petascale supercomputers. The purpose of MS-FLUKSS is to simulate flows of magnetized, partially ionized plasma experiencing collisions with neutral atoms [13], [14], [12], [8] , [7].

The suite self-consistently combines a module to solve ideal MHD equations using high-resolution, shock-capturing numerical schemes and adaptive mesh refinement among many other physical packages. Our plan is to follow the successful line of development of MS-FLUKSS joining classical ideal MHD with the AMR CHOMBO package and implement a new module in it that enables us to deal with RMHD problems. The goal is to replace the *MRGENESIS* code as provider of RMHD numerical models in cases in which an extremely high spatial resolution is needed to adequately describe the dynamics of the system.

As mentioned above, the simulation of the thermal fluid is technically independent of the evaluation of its non-thermal emission. Thus, the numerical RMHD models can be computed either with *MRGENESIS* or with the extension of MS-FLUKSS to ideal RMHD, while the observational signature can still be computed with *SPEV*. In this regard, it is worth to point out that *MRGENESIS*, *SPEV* and MS-FLUKSS store the numerical models or their respective checkpoint files in HDF5 format[5]. The reason to adopt this standard is that HDF5 is a data model, library, and file format for storing and managing data in parallel supercomputers. It supports an unlimited variety of data types, and is designed for flexible and efficient I/O and for high volume and complex data. As we shall see in Chap. 4, a number of improvements have been done in *MRGENESIS* and *SPEV* to gain extra efficiency in applications in which intensive I/O is needed.

The task of implementing RMHD modules on MS-FLUKSS is beyond the scope of this thesis, and has been performed by other members of the

[4] `https://commons.lbl.gov/display/chombo/Chombo+-+Software+for+Adaptive+Solutions+of+Partial+Differential+Equations`

[5] http://www.hdfgroup.org/HDF5/

CAMAP group of the Department of Astronomy and Astrophysics of the University of Valencia in collaboration with the group of Prof. Pogorelov at the University of Alabama in Huntsville. My main taks here has been to provide an automatic installation wizard for the complete set of libraries involved both in CHOMBO and in MS-FLUKSS, as well as to support the RAG and of the CAMAP members in any of the technical aspects related to the optimal coding in C++ and code debugging.

The methodology

The methodology used in present work is based on the scientific method. The goal of such methodology is obtaining clear and verifiable knowledge of our object of research. Particularly, our method is of the kind **analytical-experimental**. We systematically consider all the parameters and variables on which our problem depends on. The knowledge acquired after the analysis, which has to be verifiable, is used to modify these parameters and variables. This is done in order to improve models that help us understanding the problem, which in turn allows us to take new measures and refine our insights on the subject. The measures can be modifications, new implementations, etc. of our object of study. When the problem is solved, the solution must be verified with tests in all viable scenarios, changing all input parameters and variables. With these tests we can formulate a thesis or a conclusion about the performance of a new model which has been made from modifications and new implementations.

The phases of the process which achieves this goal are as follows:

1. *Projective phase*: The first stage in the methodology is to order and systematize our goals, formulate the questions and establish the current state-of-the-art as our starting point. Here we separate the pre-existing knowledge from that we aim to obtain. In this phase a proper formulation of the problem at hand is the most important goal. In Chap. 2, Sec. 2.1 we formulate the problems solved in this thesis.

2. *Methodological phase*: In this phase we need to fix the strategy of our study, formulate and check the models, change their parameters, and form different points of view. The models must be verifiable and are the basis for the next phase. This phase corresponds to the Chap. 2, Sec. 2.2.

3. *Technical phase*: Now we have to execute the procedures, modifications or implementations in the object of study, using the appropriate tools and techniques. After the implementation, we have to check again the

results and confirm that the improvements have been achieved and that the new knowledge has been acquired. Chapters 3, 4, 5 are where this is done.

4. *Phase of synthesis*: Once we have obtained all information from our study we can make conclusions about the thesis. Chap. 6 is where we provide the synthesis of the work performed in this thesis.

Chapter 1

Preparation of the work environment

1.1 Quick Start

Our local development machine is an Apple Xserve with two Quad-Core Intel Xeon Nehalem processors running at 2,93 GHz. On this server all the tools necessary for application development have been installed.

Most of the codes used by the members of the RAG are developed in Fortran 90, and a small number of post-processing tools has been written in C++. We have compiled the codes using GNU GCC. GCC stands for *GNU Compiler Collection* and includes the frontends for C, C++, Objective-C, Fortran, Java, Ada, and Go languages, as well as their corresponding standard libraries for (libstdc++, libgcj,...). More information about GCC can be found at its website `http://gcc.gnu.org/`.

We use the Message Passing Interface (MPI) standard for distributed memory parallelism. MPI implementation we use is *MPICH*, a high performance and portable implementation of this standard. For more information see `http://www.mpich.org`. At the beginning of the work in this thesis, MPI standard 3.0 was not finalized yet, and for this reason we use the MPICH2 implementation of the MPI-2 standard. The installation is a complex task since there is a large number of configuration options, and, as we will see below (Sec. 1.2), we encountered some errors during the procedure of installation.

HDF5 is a data model, library, and file format for storing and managing large amounts of information. It supports an unlimited variety of datatypes, and is designed for a flexible and efficient I/O and, especially, for large volumes and high complexity of data. HDF5 is portable and extensible, allowing applications to evolve in their use of HDF5 without losing access to data stored by older versions. One of the main reasons why we chose HDF5 is its ability to read and write data in parallel using the MPI I/O

library. The HDF5 Technology suite includes tools and applications for managing, manipulating, viewing, and analyzing data in the HDF5 format (see `http://www.hdfgroup.org/HDF5/` for more information). In the case of the RAG, these tools were installed to be used as libraries for reading and writing the output files. The the optimal data structure in these files has been studied in chapter 5.

In Sec. 1.1.1 we show details of the tasks which needed to be completed for the computing environment to be prepared and detail some of the problems encountered in Sec. 1.2. Sec. 1.3 explains how we searched for solutions and in Sec. 1.4 we give our best solution found to date.

1.1.1 Tasks done

Here we briefly explain the tasks which we needed to perform in order to install all of the tools. We give the Mac OSX shell commands for each installation under:

- GCC 4.6 with gfortran (with MacPorts):

```
$ sudo port install gcc46 +gfortran
$ sudo port list gcc46
gcc46 @4.6.3 lang/gcc46
```

- Perl 5 (with MacPorts):

```
$ sudo port list perl5
perl5 @5.12.3 lang/perl5
```

- MPICH2 v 1.4.1p1

```
../mpich2-1.4.1p1/configure --prefix=/opt/ghdf5_46 --enable-fortran
--enable-cxx --enable-fast CC=/opt/local/bin/gcc-mp-4.6
CXX=/opt/local/bin/g++-mp-4.6 FC=/opt/local/bin/gfortran-mp-4.6
F77=/opt/local/bin/gfortran-mp-4.6
```

- HDF5 1.8.6
  - Configuration with compatibility flags CPPFLAGS=DH5_USE_16_API and –enable-production

```
../hdf5-1.8.6/configure --prefix=/opt/ghdf5_46
FC=/opt/ghdf5_46/bin/mpif90 F77=/opt/ghdf5_46/bin/mpif77
CC=/opt/ghdf5_46/bin/mpicc CXX=/opt/ghdf5_46/bin/mpic++
--enable-fortran --enable-parallel --enable-production
CPPFLAGS="-DH5_USE_16_API"
```

  - In order to use the parallel read and write we have to configure HDF5 with the option –enable-parallel. That flag produces the compilation of the parallel libraries.
  - The compiler and tools installed are located in /opt/ghdf5_46.

- GNU make

```
GNU Make 3.82
Built for x86_64-apple-darwin10.5.0
Copyright (C) 2010 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later
<http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and
redistribute it.
There is NO WARRANTY, to the extent permitted by law.
```

1.2 Problems encountered during installations

Here we give information about some of the problems encountered during the MacPorts selfupdate. Using a cron task we perform periodic self-updates of MacPorts library. Initially this worked, but at some point an update affected the installation of gcc45 and gcc46 compilers. The problems encountered were:

- The compilation did not work because of a link error for standard library:

```
ld: library not found for -lstdc++
```

- The library is not found in any of the expected paths:

```
/opt/local/lib/gcc46/lib
/opt/local/lib
/urs/lib
```

1.3 A search for solutions

Here we list the steps that were taken in order to solve the problem.

1. *Attempt*: reinstall gcc46 using MacPorts. *Result*: the problem persists.

2. *Attempt:* uninstall gcc46 and install gcc45 using MacPorts. *Result*: the problem persists.

3. *Attempt:* uninstall all compilers and install gcc48 using MacPorts. *Result*: the problem persists.

4. *Attempt*: download the compiler source from GCC website; download the source code of the dependencies and install them; compile the GCC using the default compiler installed in OSX (Xcode version 3.2). *Result*: the error:

```
 Undefined symbols for architecture x86_64:
 "_libiconv_open", referenced from:
 identifier_to_locale(char const*) in libcommon.a(pretty-print.o)
 "_libiconv", referenced from:
 identifier_to_locale(char const*) in libcommon.a(pretty-print.o)
 "_libiconv_close", referenced from:
 identifier_to_locale(char const*) in libcommon.a(pretty-print.o)
ld: symbol(s) not found for architecture x86_64
```

   However the library was correctly installed and the problem remains, for this reason we continue searching the solution.

5. *Attempt*: revise GCC dependencies. In the official web page of GNU, there are a lot of dependencies between packages that need the compiler to be installed successfully. These packages were installed with MacPorts.
   Installation of ports:

```
sudo port install libcompat
sudo port install libtool
sudo port install pkg-config
sudo port install pmk
sudo port install libconfig-hr
sudo port install libconfuse
sudo port install perl5.16
sudo port install isl
sudo port install cloog
sudo port install m4
sudo port install gmake
sudo port install gettext
sudo port install gperf
```

```
sudo port install dejagnu
sudo port install expect
sudo port install autogen
sudo port install flex
sudo port install texinfo
sudo port install tex
sudo port install diffutils
```

Result: The compilation still fails because references were not found when building GCC.

6. *Attempt*: Set the LD_LIBRARY_PATH environment variable to the path where the dynamic library libstdc++ is located

```
export LD_LIBRARY_PATH=/usr/local/lib/
```

Result: The problem persists with the error of bad references in a large number of files.

7. *Attempt*: Installing gcc48 with MacPorts version of libstdc++. *Result* successful installation using MacPorts:

```
sudo port install libstdcxx
sudo port -f activate libstdcxx-devel
sudo port install gcc48 +gfortran
sudo port select gcc mp-gcc48
```

8. MPICH2 was configured and compiled without any problem:

```
../mpich2-1.3.2p1/configure --prefix=/usr/local
--enable-fortran --enable-cxx --enable-fast
--with-pm=hydra --with-device=ch3:nemesis
CC=/opt/local/bin/gcc-mp-4.8
CXX=/opt/local/bin/g++-mp-4.8
FC=/opt/local/bin/gfortran-mp-4.8
F77=/opt/local/bin/gfortran-mp-4.8
```

9. HDF5 1.8.9 was configured:

```
../hdf5-1.8.9/configure --prefix=/opt/ghdf5_46_chombo
FC=/usr/local/bin/mpif90 F77=/usr/local/bin/mpif77
CC=/usr/local/bin/mpicc CXX=/usr/local/bin/mpic++
--enable-fortran --enable-parallel
--enable-production CPPFLAGS=-DH5_USE_16_API
```

In compilation, we got an error in file hdf5-1.8.9\test\err_compat.c

```
too many arguments to function H5Eset_auto1.
```

We fixed it and then HDF5 compiled successfully.

10. We try to compile CHOMBO, but we get several errors:

```
error: 'makeItSoEnd' was not declared in this
scope, and no declarations were found by
argument-dependent lookup at the point of
instantiation [-fpermissive]
makeItSoEnd(*this, this->interval());
```

It can be fixed by adding the flag -fpermisive in make.defs.GNU located at
\chombo\lib\mk\compiler

```
 _cxxbaseflags := -fpermissive -Wno-long-long
-Wno-sign-compare  -Wno-deprecated
-ftemplate-depth-25
```

11. MS-KFLUSS is compiled with several warnings related to this error. The compilation finished successfully, but at runtime we get incorrect variable values (including NaNs).

1.4 Best solution

Due to the magnitude of the problems, we decided to uninstall all compilers and went back to GCC 4.5 with MPICH2 1.3.2p1 and HDF5 1.8.6 (1.8.7 and higher produce compatibility problems due to deprecated functions).

Since to the MacPorts uninstall command does not clean all the directories, we deleted all folders belonging to the installations of gcc46, gcc47 and gcc8. We also had to delete the libraries libstdc++ and the soft links created in the folder

```
/usr/local/lib
```

.

1.4.1 GCC45 installation

```
cd /opt/local/lib
sudo port uninstall libstdcxx
sudo rm libstdc++.6.dylib*
sudo port uninstall libstdcxx-devel
sudo port uninstall gcc48
sudo port uninstall libstdcxx-devel
sudo port uninstall libstdcxx
sudo port install gcc45 +gfortran
sudo port select gcc gcc45
```

1.4.2 MPICH2-1.3.2P1 installation

```
cd /opt/install/mpi_chombo_install
sudo /usr/local/sbin/mpeuninstall
sudo make -j clean
sudo ../mpich2-1.3.2p1/configure --prefix=/usr/local
--enable-fortran --enable-cxx --enable-fast
--with-pm=hydra  --with-device=ch3:nemesis
CC=/opt/local/bin/gcc-mp-4.5
CXX=/opt/local/bin/g++-mp-4.5
FC=/opt/local/bin/gfortran-mp-4.5
F77=/opt/local/bin/gfortran-mp-4.5
sudo make -j
sudo make -j install
```

1.4.3 HDF5 1.8.6 installation

```
cd /opt/install/hdf_chombo_install
sudo ../hdf5-1.8.6/configure --prefix=/opt/ghdf5_45_chombo
FC=/usr/local/bin/mpif90 F77=/usr/local/bin/mpif77
CC=/usr/local/bin/mpicc CXX=/usr/local/bin/mpic++
--enable-fortran --enable-parallel --enable-production
CPPFLAGS=-DH5_USE_16_API
sudo make -j
sudo make -j install
```

1.4.4 Modification of MacPorts self-update script

The MacPorts daily self-update script is executed daily to automatically update the packages installed by MacPorts. The periodic task is implemented as an OSX Launch Daemon which calls a shell script. The script was modified to include an exception so that GCC compilers are not updated automatically, but will be updated by hand. The current version of the shell script which is called by the daemon is:

```
#!/bin/sh
##
# Update packages of MacPorts
##
datenow=‘date +%m/%d/%Y-%H:%M‘
LOGPATH=/Library/UpdatePortsDaemon
echo "$datenow - MacPorts Update" >> $LOGPATH/out.log
sudo /opt/local/bin/port selfupdate >> $LOGPATH/out.log
sudo rm $LOGPATH/outdated.out
sudo rm $LOGPATH/outdated_mod.out
sudo rm $LOGPATH/outdated.sh
sudo /opt/local/bin/port outdated > $LOGPATH/outdated.out
sudo grep "<" outdated.out > $LOGPATH/outdated_mod.out
gccport="gcc"
while IFS=$’ ’ read portoutdated version
do
  if test "${gccport:0:3}" != "${portoutdated:0:3}"
  then
     sudo echo "sudo /opt/local/bin/port upgrade
     $portoutdated" >> $LOGPATH/outdated.sh
     sudo /opt/local/bin/port upgrade
     $portoutdated  >> $LOGPATH/out.log
  fi
done < outdated_mod.out
sudo /opt/local/bin/port uninstall
inactive >>$LOGPATH/out.log
echo "$datenow - End" >> $LOGPATH/out.log
```

1.4.5 Implementation of a script of full installation

After the full installation has been performed, we have implemented a shell script that executes all tasks necessary for automatic installation of the com-

pilers and the CHOMBO library. This installation can be executed on computers of the Group and the University of Valencia, including *Lluis Vives* cluster. [1]

We have created an archive containing the script and the necessary compilers. The versions of the compilers and wrappers that are installed by the script correspond to the versions that have been proven to work correctly (see Sec. 1.4). The contents of the archive are:

- *MPICH2* source
- *HDF5* source
- *CHOMBO* source
- Installation script

In the CHOMBO folder there is a template which configures the installation of this library. We have modified several configuration variables to enable the correct installation by this template. The modified variables are: hostname, FC, MPICXX, hdf_dir and MAC. We have substituted their values \$1, \$2, \$3, \$4 and \$5, in order for the installation script to be able to pass them as arguments to the template. This way we can always insert the correct values, depending on the local installation.

The script is as follows:

```
#!/bin/sh

hostname=‘hostname‘
if [ $hostname = "vives" ]
then
  module load cc_intel
  module load fc_intel
fi
current=‘pwd‘

# Untar mpich2
tar xvf mpich2.tar.gz

# Untar hdf5
tar xvf hdf5.tar.gz

# Untar chombo
tar xvf chombo.tar chombo

# Make installation folders
mkdir mpi_chombo_install
mkdir hdf_chombo_install
```

[1] Cluster located at Servei d'Informatica of the University of Valencia. Altix Ultraviolet 1000 with 30 CPU Xeon 7500 hexacore at 2,67GHz, 18MB L3 cache memory on-die, 960 GB of RAM. http://www.uv.es/lluisvives

```
mkdir ghdf_chombo_install
ghdf=$current/ghdf_chombo_install

### MPICH2 installation
gcc=‘which gcc‘
gxx=‘which g++‘
gfortran=‘which gfortran‘
cd $current/mpi_chombo_install
../mpich2/configure --prefix=$ghdf --enable-fortran
--enable-cxx --enable-fast --with-pm=hydra
--with-device=ch3:nemesis CC=$gcc CXX=$gxx FC=$gfortran F77=$gfortran
make
make install

### HDF5 installation
cd $current/hdf_chombo_install
../hdf5/configure --prefix=$ghdf FC=$ghdf/bin/mpif90
F77=$ghdf/bin/mpif77 CC=$ghdf/bin/mpicc CXX=$ghdf/bin/mpic++
 --enable-fortran --enable-parallel --enable-production
CPPFLAGS=-DH5_USE_16_API
make
make install

### Chombo configuration and compilation
cd $current/chombo/lib/mk/local
MakeDefsLocal="$current/chombo/lib/mk/local/Make.defs.$hostname"
TEMPLATE="$current/chombo/lib/mk/local/Make.defs.$hostname.template"

MakeDefsLocal="Make.defs.$hostname"
TEMPLATE="Make.defs.template"
echo $MakeDefsLocal
echo $TEMPLATE

mpif90="$ghdf/bin/mpif90"
mpicxx="$ghdf/bin/mpic++"

system=‘uname -s‘
mac=’false’
if [ $system = "Darwin" ]
then
  mac=’true’
fi

sed ’s;$1;’"$hostname"’;
s;$2;’"$mpif90"’;
s;$3;’"$mpicxx"’;
s;$4;’"$ghdf"’;
s;$5;’"$mac"’;’ <$TEMPLATE> $MakeDefsLocal

cp $MakeDefsLocal $current/chombo/lib/mk/Make.defs.local
cd $current/chombo/lib
make lib
```

Chapter 2

Profiling and analysis of the computational aspect of SPEV

In this chapter we discuss the basic algorithm of the *SPEV* code and perform profiling of its two main components (preprocessor and postprocessor). In Sec. 2.1 we describe the *SPEV* algorithm and give reasons for its optimization. Subsequently, in Sec. 2.2 we perform extensive profiling and testing of various aspects of the *SPEV* code.

2.1 Motivation for code optimization

The numerical code *SPEV* [5] is written in Fortran 95, parallelized using OpenMP and uses HDF5 library for I/O. Fig. 2.1 shows a typical state of execution of *SPEV* postprocessor (see Section 2.1.1) on a production machine. In the example *SPEV* uses 48 CPUs and more than 512GB of RAM, which is around 25% of the total memory on the machine. At the moment the snapshot was taken the code has been running for approximately 8 hours, and the calculation was half-done. *SPEV* allocates memory dynamically as needed and the 512GB in the example might turn out to be much more by the time the calculation is finished. Furthermore, the total amount of needed memory cannot be precisely known in advance. This has on occasions caused the machine to run out of memory and to start paging, which had a disasterous consequences for performance. Therefore, it was decided to optimize the memory usage and to reduce it as much as possible (see Chapter 4).

Figure 2.1: Memory and CPU consumption of a typical execution of the *SPEV* postprocessor (see Section 2.1.1) on the *LluisVives* cluster at the University of Valencia. The figure is a screenshot of the output of the *htop* command. See text for more details.

Apart from the memory usage, *SPEV* also requires the storage of rather large files, which are produced by processing the results of the RMHD simulations. A calculation in Fig. 2.1 requires a relatively small input file whose size is approximately 50 GB. However, there can be cases where the size of the input file reacher hundreds of GB, or even TB. It is for this reason we decided improving the usage of disk space as well, since often the input files need to be copied over the network from the local clusters of the RAG to a supercomputer where the production runs will be performed.

2.1.1 *SPEV* algorithm and its profiling

The goal of a *SPEV* calculation is to compute the time- and frequency-dependent image from a collection of snapshots produced by *MRGENESIS*. In the simplest terms the algorithm can be written as:

1. Define the *virtual detector* where the simulated radiation from the non-thermal particles will be observed.
2. For a snapshot of *MRGENESIS* simulation:
   (a) Compute where non-thermal particles are injected into the flow and calculate their initial energy distribution.
   (b) Transport non-thermal particles and calculate their radiation and adiabatic losses or gains.
   (c) Calculate the contribution of the current distribution of non-thermal particles to the virtual detector.
3. If there are more snapshots left, go to 2.
4. Compute the final image on a virtual detector by solving the radiative transfer equation for each of its *pixels*.

This algorithm processes all the snapshots of an RMHD simulation in sequence and keeps the contribution from each one of them to the final image in working memory until the last file has been processed, and only then can

the final image be computed. This is the reason for the extensive memory usage.

The step 1 of the algorithm defines the orientation, size, time and frequency of observation of the object simulated by *MRGENESIS*. However, steps 2.(a) and 2.(b) of the algorithm are independent of those parameters, and the repetition of those steps can be avoided if their results are stored in an intermediate file. The modified algorithm is:

1. **Preprocessing:**
   (a) Open the preprocessed output file.
   (b) For a snapshot of *MRGENESIS* simulation:
      i. Compute where non-thermal particles are injected into the flow and calculate their initial energy distribution.
      ii. Transport non-thermal particles and calculate their radiation and adiabatic losses or gains.
      iii. Store the current positions and energy distribution of each non-thermal particle in the preprocessed file. The output file is structured into *blocks*.
   (c) If there are more snapshots left, go to ii.
2. **Postprocessing:**
   (a) Define the *virtual detector* where the simulated radiation from the non-thermal particles will be observed.
   (b) For a block of the preprocessed file calculate the contribution of the current distribution of non-thermal particles to the virtual detector.
   (c) If there are more blocks left, go to (b).

It should be noted that preprocessing only needs to be done once, while the postprocessing can be repeated many times, for different size and resolution of the virtual detector, or for different orientation, observation frequencies and times.

The output of the preprocessing can be a very large file. The postprocessing may require a large amount of RAM. Therefore, we optimize both of these processes according to the resource which each one of them needs most.

In general terms, our aim is to study the performance of *SPEV*, and for this reason we have performed a profiling of the application. This gave us information to locate the most expensive routines of the code, and we have focused in these routines of *SPEV*. The best configuration will be based on optimizing according to these parameters:

- Speed of execution (preprocessing and postprocessing)

- Memory consumption (postprocessing)
- Disk usage (preprocessing)
- Bandwidth requirements (preprocessing and postprocessing)

We must remember that the stage at which to execute this application, the resources are limited and shared by other users: memory, disk space, number of CPUs, time of CPU usage. Also, the supercomputer and the local cluster are not in the same physical location, so the time spent copying the preprocessed files also is a factor to consider with regards the total time of execution.

2.1.2 *MRGENESIS* and *SPEV* parallelization

In this section we give details about the parallelization of *MRGENESIS* and *SPEV*

HYBRID PARALLELIZATION OF *MRGENESIS*

MRGENESIS is a code implemented in Fortran 90. The first parallel version of the code [10] was parallelized using Message Passing Interface (MPI), and intended to be run on distributed memory architectures. MPI implementation we use is MPICH, a high performance and portable implementation of this standard[1]. Due to the ever increasing resolution requirements and the consequent increase in the number of processors necessary to run simulations, this code was upgraded to a hybrid model of parallelization. This model combines the distributed and shared memory parallelism models. The main motivation for this upgrade was the intent to improve the *MRGENESIS* performance on the MareNostrum Supercomputer. At that time, MareNostrum[2] had a hybrid memory architecture: memory was shared by all processors inside blades (each Server Blade JS21 had two dual-core processors PowerPC 970MP) and the blades were interconnected via Mirynet network sharing a distributed memory.

We performed the upgrade of *MRGENESIS* [11], [2] so that it uses OpenMP on a blade, and MPI between the blades. The upgrade was successful and the result was that the hybrid code scales up to 7200 processors (see Fig. 2.2).

[1] `http://www.mpich.org`
[2] `http://www.bsc.es/marenostrum-support-services/marenostrum-system-architecture`

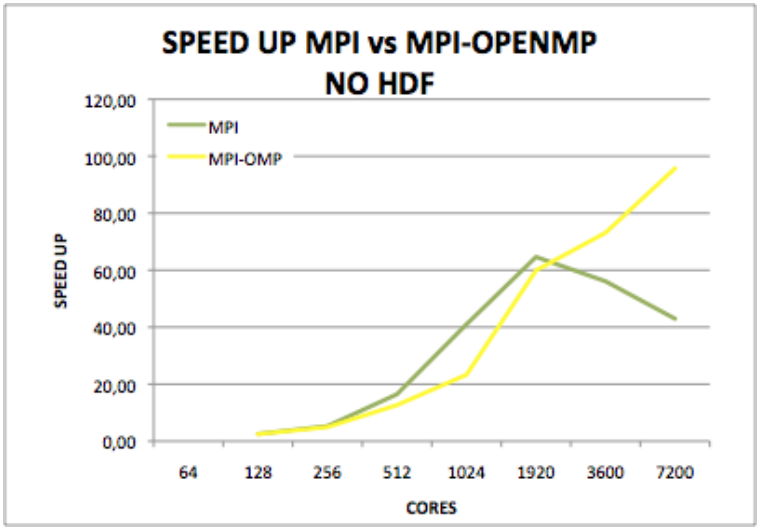

Figure 2.2: Code scaling for a standard hydrodynamic problem in astrophysics (collision of stellar winds) using a different numbers of processors and combinations of MPI and OpenMP threads on MareNostrum II Supercomputer. Numerical grid resolution was 8640×8640. The code was compiled using IBM XLF90 compiler with the following options: -O3 -qstrict -q64 -qtune=ppc970 -qarch=ppc970 -qcache=auto. The number of iterations was 600. The speed-up was measured relative to a test with 64 CPUs

The code upgrade was completed over several months with the collaboration of the Barcelona Supercomputing Center, who kindly allowed us the exclusive use of the machine during three days in order to perform all necessary scalability tests. This work has resulted in a poster [1] which was presented during the Poster Session of 2011 PRACE Summer School[3].

PARALLELIZATION OF *SPEV* USING OPENMP

The current version of *SPEV* is parallelized using OpenMP. The loops in which the parallelism is implemented go over the non-thermal particles, which are the sources of the observed emission. The methodology used to parallelize the loop is dynamic scheduling: the iterations (particles) are distributed in chunks to threads as long as these become available to take new work. Each thread executes a chunk of iterations, then requests another, until no more work remain to be distributed.

In this way, each thread computes the contribution from its chunk of particles and updates the global arrays (virtual detector pixels, see Sec. 4.1). Since it is possible that two or more threads attempt to update the same location in the global array, we use locks to serialize the memory updates. However, this does not happen frequently and does not in fact affect the

[3] `http://www.csc.fi/english/csc/courses/archive/prace-summer-school`

parallel performance, which has been found to scale up to 114 cores of Lluis Vives cluster [4]. The reason for this is that most of the work done on the particles is in computing their emission properties and determining which pixels need to be updated, and these operations do not modify the global variables. In Sec. 4.2.2 we will explain the implementation of the new data structures which are also used in parallel.

2.2 Profiling and testing

In this section we perform profiling and testing of the *SPEV* preprocessor (Sec. 2.2.1 and 2.2.2, respectively). Afterward, and related to the results of the study of the preprocessor, we perform further tests on the postprocessor (Sec. 2.2.3).

2.2.1 Profiling the preprocessor

As mentioned in a previous section, the preprocessed file is structured into blocks. Each block contains the information about the position, velocity and energy distribution of non-thermal particles. The reason why the file is divided into blocks is because during the postprocessing phase we can choose to read only a subset of blocks.[5] The block size in the following refers to the number of particles in each block. Since we use HDF5 as the format for the preprocessed files, we have an option of compressing each block before writing it to disk. This is desirable since there are redundancies in the data and we can typically achieve a saving in disk space up to a factor of 2. Therefore, in the following sections we separate the execution time of the DEFLATE algorithm from the rest of the code.

While the performance of the preprocessing code will inevitably vary with the number, size and the data in the snapshots produced by *MRGENESIS*, we have created as representative set of snapshots as possible and performed a series of tests. In the Appendix B we shown the sampling tests done to obtain the data for the charts 2.3, 2.4 and 2.5. In the following sections we discuss the results of these tests.

[4] Cluster located at Servei d'Informatica of the University of Valencia. Altix Ultraviolet 1000 with 30 CPU Xeon 7500 hexacore at 2,67GHz, 18MB L3 cache memory on-die, 960 GB of RAM. `http://www.uv.es/lluisvives`

[5] Due to the fact that the preprocessing algorithm processes the files produced by *MRGENESIS* ordered in time, the first blocks of the preprocessed file contain particles for early evolutionary times, while the late blocks contain particles at later evolutionary stages. Therefore, when we are sure that not all the information will be needed to compute the image in a specific virtual detector configuration (e.g., in case we only need early times), not all the blocks need actually be read. This leads to significant savings of the computational resources.

FIXED BLOCK SIZE 50000

This is the first case studied, and is the block size typically used in real executions of *SPEV*. Our goal is to determine the most expensive routine and the impact of the compression level of the DEFLATE algorithm on the overall performance. The compression level is set by calling the *h5pset_deflate_f* subroutine of the HDF5 library, and can take integer values from 1 (lowest) to 9 (highest compression level).

The following table shows the percentage of execution time used in the routine DEFLATE and in the rest of the routines for the compression level from 1 to 9. The same is graphically represented in the Fig. 2.3.

COMPRESSION LEVEL	DEFLATE % Time	THPRSPEV2 % Time
1	75,9	19,2
2	75,5	18,9
3	75,5	18,9
4	75	20
5	74,8	19
6	74,5	19,5
7	75,4	19
8	75,1	19,4
9	74,8	19,5

As we can see from Fig. 2.3, the code spends approximately 75% of the time in the DEFLATE. Secondly, we see that the level of compression almost does not affect the percentage. What we can conclude from the results so far is that the block size of 50000 bytes is too big and is lost too much execution time trying to compress files with very large blocks. Also, we do not know if the same applies to other block sizes, so we need to perform an additional test with a fixed compression level, but with varying block size.

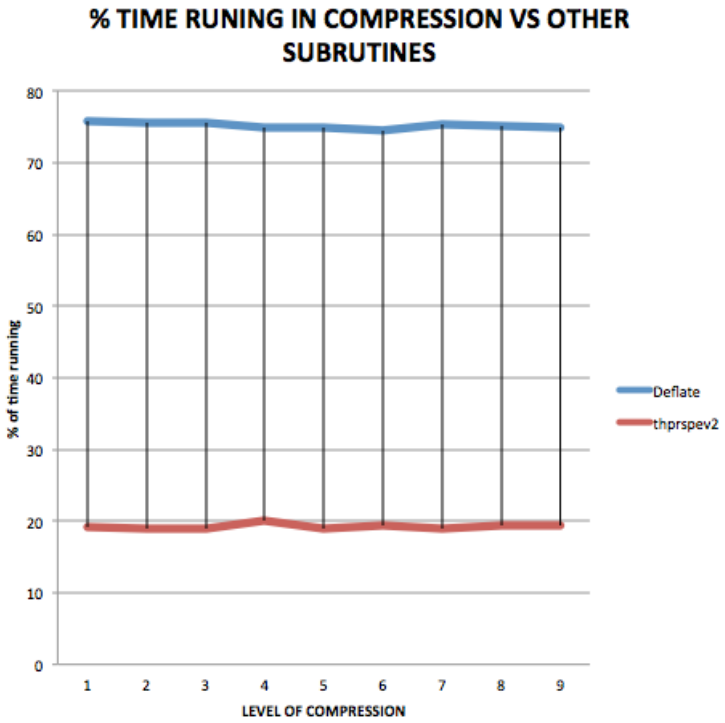

Figure 2.3: Percentage of the execution time spent on compression (blue line) and in the rest of the code (red line) as a function of the level of compression.

FIXED COMPRESSION LEVEL 6

We performed preprocessing profile analysis for different block sizes for a level of compression fixed to 6. The objective of this test is to find the optimal block size with an intermediate level of compression. In Fig. 2.4 we can see the result.

We observe in Fig. 2.4 that the routine DEFLATE takes between 40 to 80% of execution time compared to other routines (grouped under the name thprspev2) for different block sizes. From this chart we can conclude that the optimal case (where there is the minimal ratio between the time taken to compress and time taken in the execution of other subroutines) happens with a size block of 5000 bytes. In the figure this is the case in which red and blue lines are the closest.

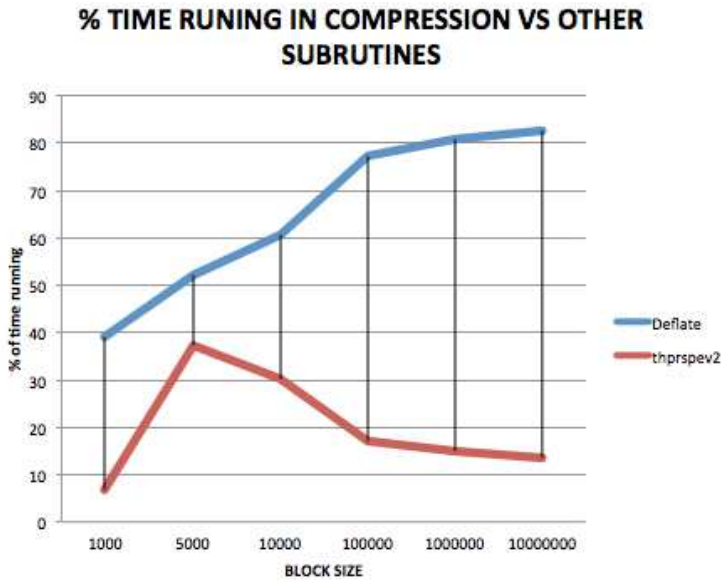

Figure 2.4: Fixed Compression Level in 6

FIXED BLOCK SIZE 5000

Finally, we check the behavior of the code for a block size fixed to 5000, which gave the best results in Fig. 2.4. As in Sec. 2.2.1 we run tests for different levels of compression, and show the results in the following table and in Fig. 2.5.

COMPRESSION LEVEL	DEFLATE % Time	THPRSPEV2 % Time
1	33,7	57,2
2	36,7	56,6
3	33,3	59,9
4	33,2	57,1
5	32,7	57,5
6	33,1	57,1
7	33,1	57,1
8	32,9	57,4
9	33	57,1

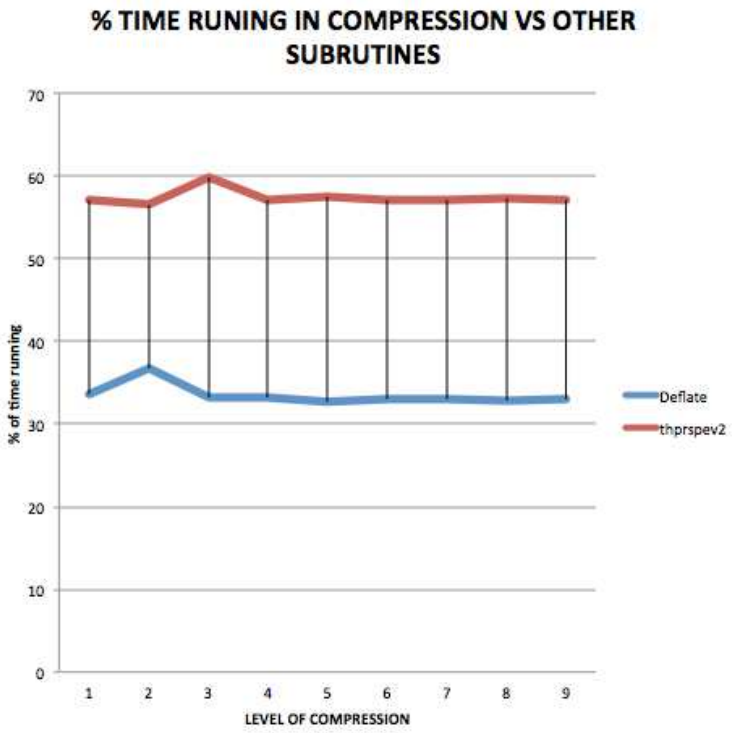

Figure 2.5: Same as Fig. 2.3, but for the block size fixed to 5000.

As can be seen, we continue obtaining that the level of compression has little effect on the percentages of routines runtime. For this reason, we decided to continue checking what is the best option in terms of levels of compression and block size, because we suspect that a block size so small could penalize the performance of the postprocessing phase of *SPEV* due to the overheads when decompressing the blocks.

2.2.2 Testing the preprocessing

In this section we perform tests of the code execution times and output file size as a function of the parameters which were determined to be important (block size and the level of compression). First we show the results for the execution time, and then for the output file size. As an input for the tests we have in total 499 snapshots produced by *MRGENESIS*.

EXECUTION TIME: VARIABLE COMPRESSION LEVEL AND BLOCK SIZE

In this test we are interested in the execution time as a function of the compression level and the block size. In order to obtain reliable results we

have repeated each test several times and taken an average execution time of all runs as a result for each test.

In the following we group the tests according to the number of snapshots used as an input for the preprocessor (as we expect and can see from the figures, the execution time strongly depends on the number of input files). We note here that the execution time does not depend linearly on the number of files. Instead, it depends almost quadratically on it. The reason is that, for each new snapshot, the *SPEV* preprocessor has to process all the non-thermal particles injected in the previous iterations before injecting new particles. This means that the amount of work grows disproportionally with the number of input files.

The charts show the execution times for three different block sizes, and for three different compression levels (plus a run with no compression, indicated as "Level 0" in the figures) for each block size.

1. Test with 198 input files:
   The Fig. 2.6 shows how the execution time increases the larger the block size is (as long as we use compression). If we do not use compression we obtain lower execution times regardless of the block size.

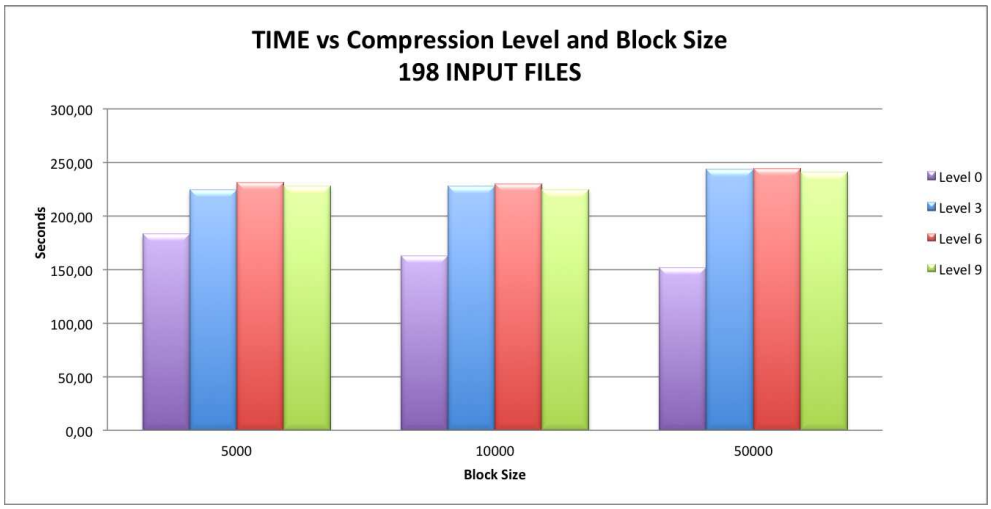

Figure 2.6: Times as a function of block size and compression level for 198 input files.

2. Test with 298 input files:
   As discussed above, using more files means that *SPEV* has to process more non-thermal particles than is the case when we use 198 input files. This means that the percentage of time spent compressing and writing the blocks into the output file decreases. We can see this in Fig. 2.7, where the execution times is independent of the compression level and the block size, except for the block size 50000, where the runs without

compression still run significantly faster.

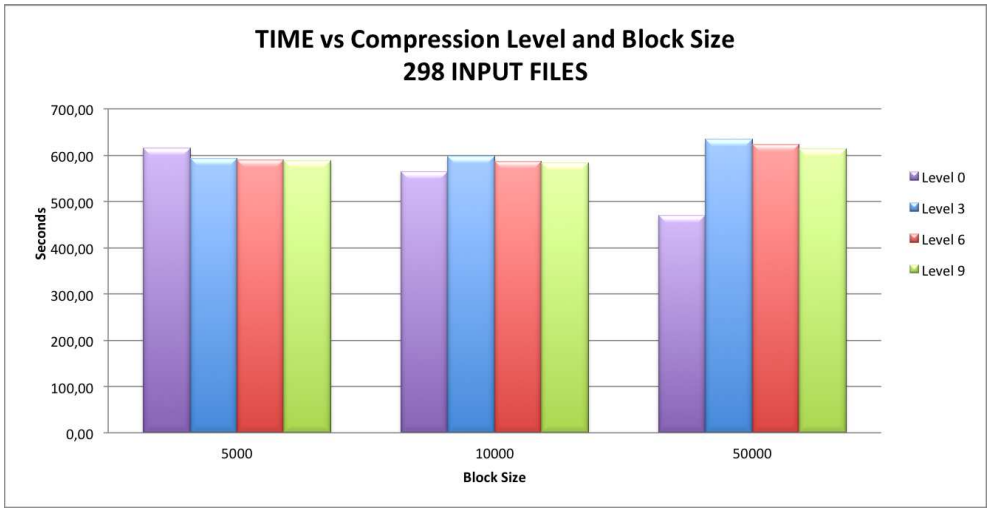

Figure 2.7: Times as a function of block size and compression level for 298 input files.

3. Test with 398 input files:
   The results of this test are very similar to the previous case with 298 files. Fig. 2.8 shows that now the results for the block size 5000 and 10000 are even closer than in Fig. 2.7.

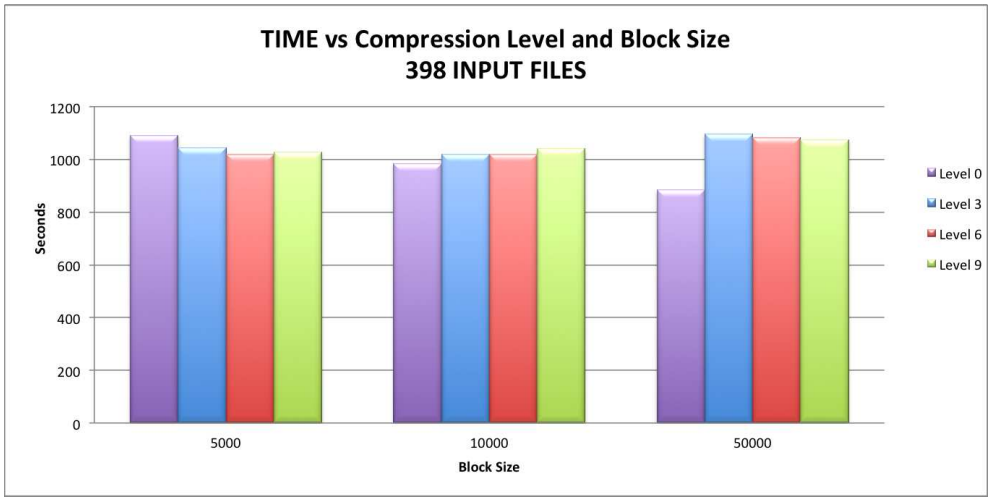

Figure 2.8: Times as a function of block size and compression level for 398 input files.

4. Test with 498 input files:
   This case, while similar to the previous two, shows in Fig. 2.9 that it is best to use the block size 10000 for any level of compression.

What we can conclude is that the number of input files is the predominant

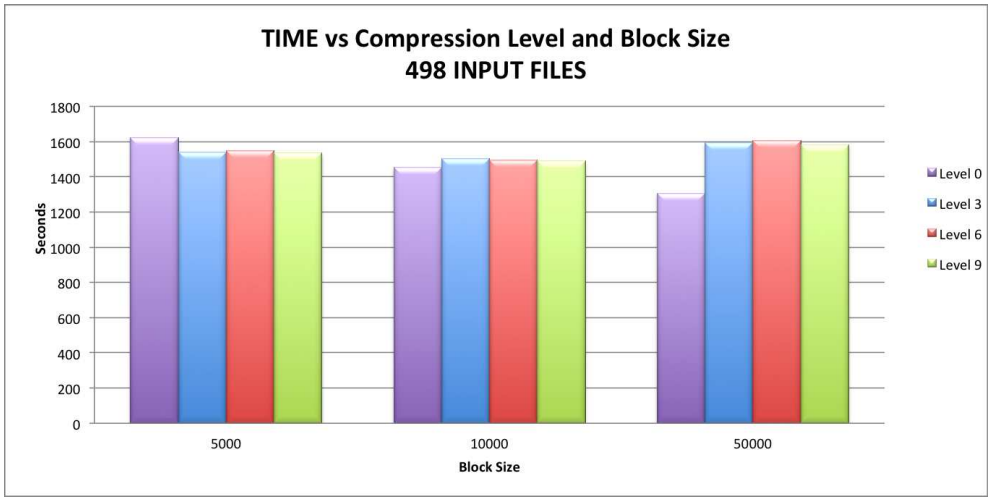

Figure 2.9: Times as a function of block size and compression level for 498 input files.

factor in determining the execution time. The next one in importance is the question whether the compression is used or not. In the next section we will study in more details the dependence on the compression level.

EXECUTION TIME: VARIABLE NUMBER OF INPUT FILES AND BLOCK SIZE

In the previous section we determined that the number of input files has the biggest influence on the execution time. In this section we present the results of our tests as a function of the number of input files and the block size, while grouping the results according to the compression level.

1. Compression level 3:
   The Fig. 2.10 shows that the time increases approximately quadratically with the number of files, regardless of the compression level. The fastest execution is obtained for the block size 10000.

2. Compression level 6:
   The results for this compression level, seen in Fig. 2.11 are very similar to those for the compression level 3.

3. Compression level 9:
   This case, shown in Fig. 2.12 is similar to the previous two, but the most significan improvement is seen with 498 input files and a block size of 10000 bytes.

From this section we conclude that the best results are obtained with the block size 10000. In the next sections we study the size of the output files.

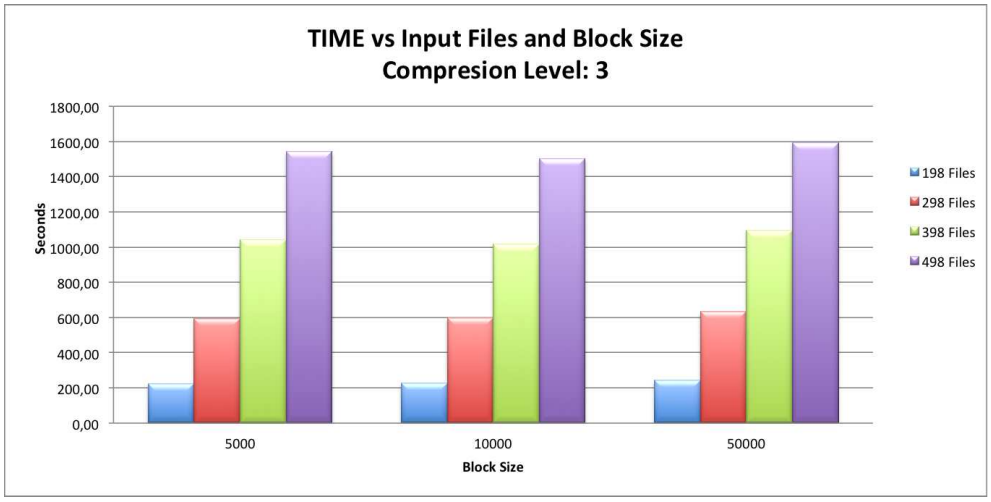

Figure 2.10: Times as a function of block size and number of files for compression level 3.

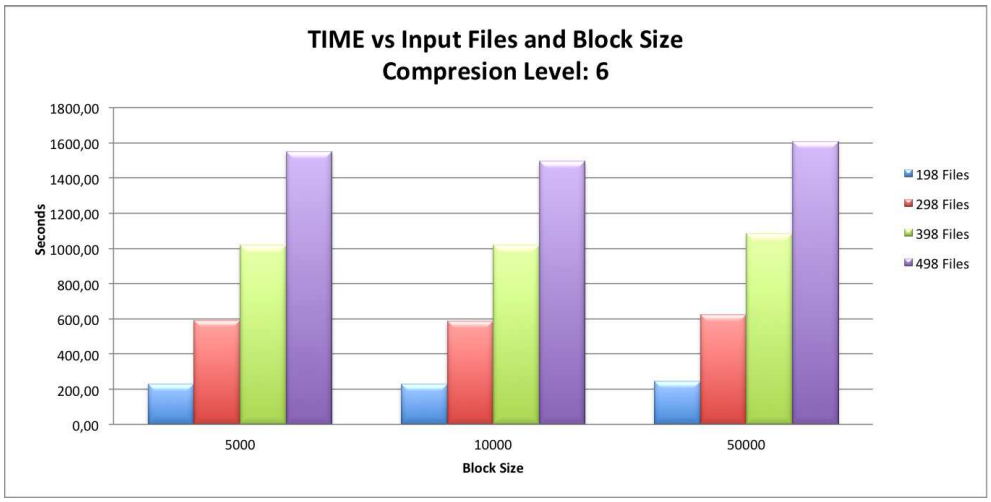

Figure 2.11: Times as a function of block size and number of files for compression level 6.

OUTPUT FILE SIZE: VARIABLE COMPRESSION LEVEL AND BLOCK SIZE

The motivation to minimize the output file size is, on the one hand, to reduce the use of the disk space. On the other hand, it is to reduce the bandwidth needed to transfer the files to and from the production and local machines. In this section we show the output file size as a function of the compression level and the block size. The results are grouped by the number of input files processed.

1. Test with 198 input files:
   In Fig. 2.13 we see that the total file size decreases by more than a factor of two once the compression is used. Analyzing in more detail we also

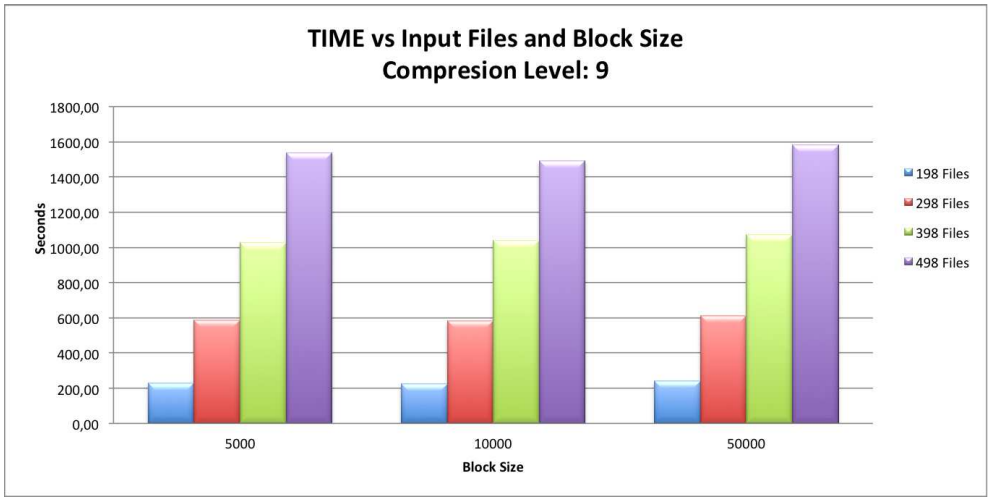

Figure 2.12: Times as a function of block size and number of files for compression level 9.

see that the file size decreases by approximately 15% when increasing the compression level from 5000 to 50000. These results are independent of the block size.

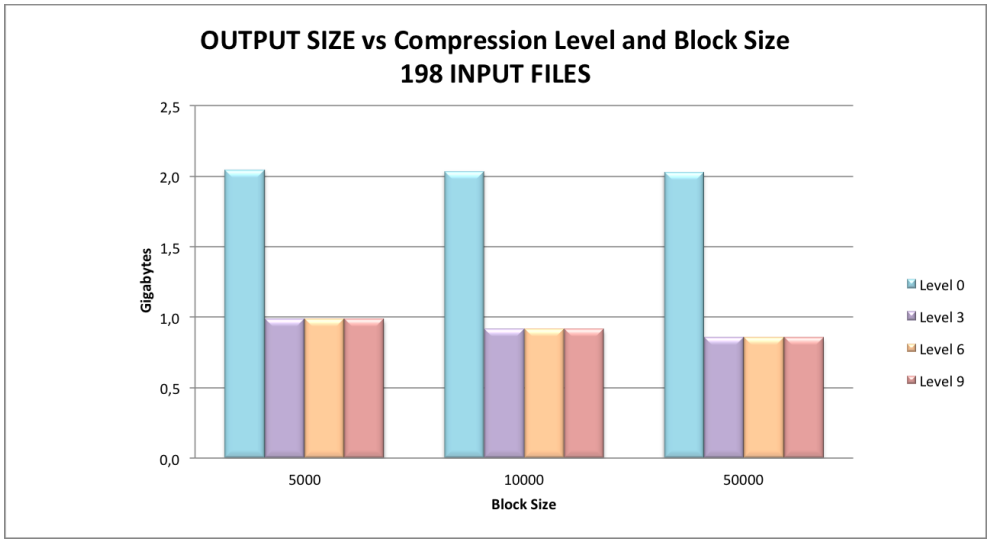

Figure 2.13: Size as a function of block size and compression level for 198 input files

2. Test 298 input files:
   The results of this test, shown in Fig. 2.14, are very similar to the case with 198 input files, but here we also see that larger block sizes decrease the compressed file size more than the smaller block sizes, a 15% less.

3. Test with 398 input files:

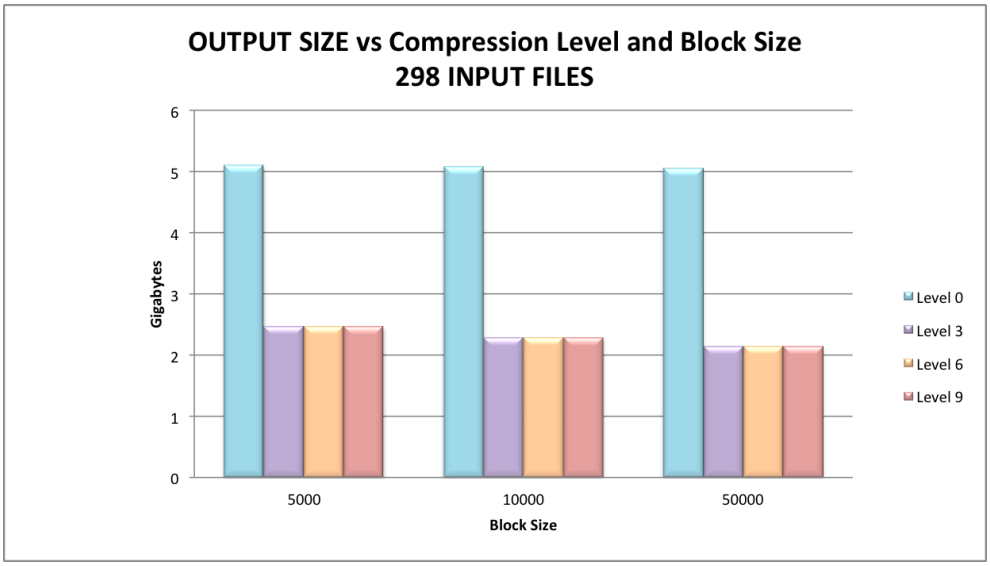

Figure 2.14: Size as a function of block size and compression level for 298 input files

In Fig. 2.15 we see a similar behaviour to the previous test.

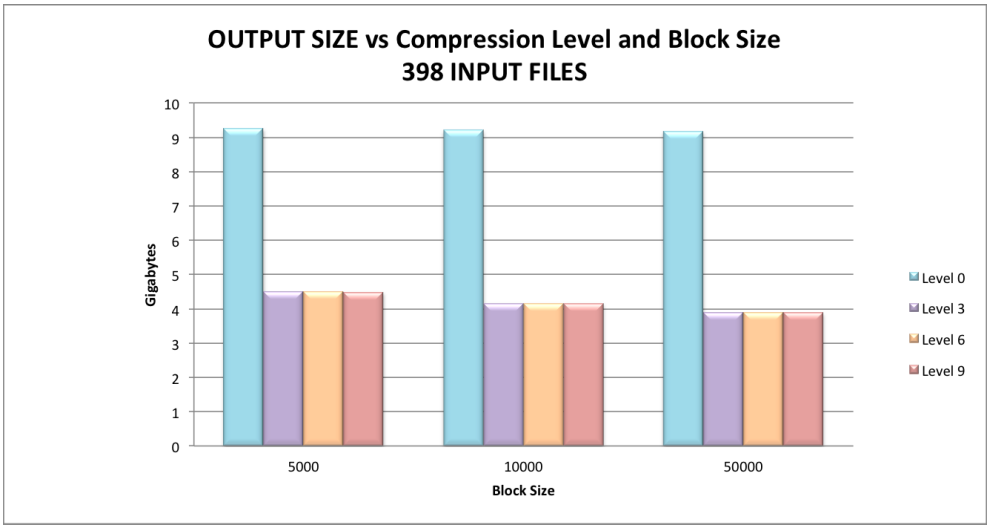

Figure 2.15: Size as a function of block size and compression level for 398 input files

4. Test with 498 input files:
   The final test, whose results are shown in Fig. 2.16, show the same behaviour as the others.

The conclusion of this section is that using any compression level significantly reduces the file size. The exact value compression level plays a moderate role in reducing the file size.

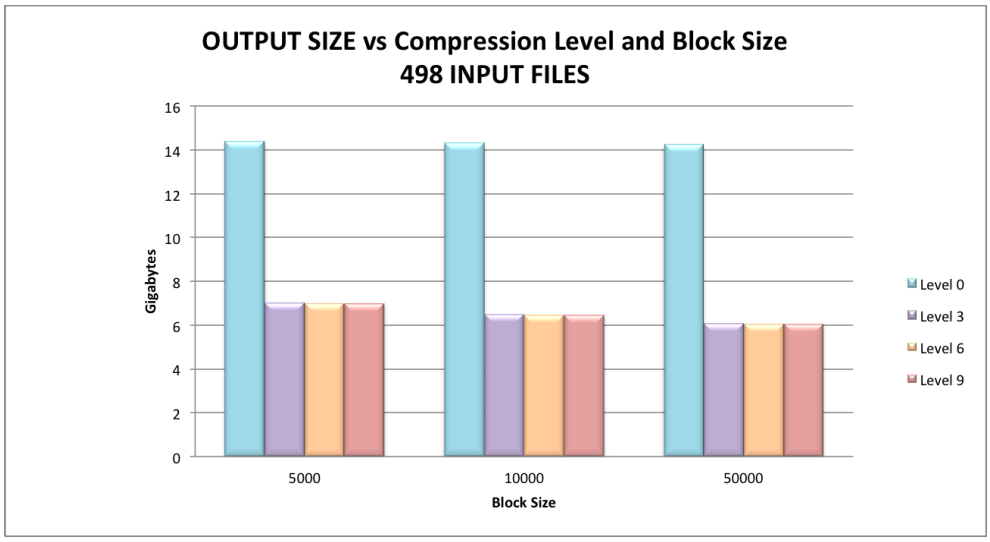

Figure 2.16: Size as a function of block size and compression level for 498 input files

OUTPUT FILE SIZE: VARIABLE NUMBER OF INPUT FILES AND BLOCK SIZE

In analogy to the first two sections, in this section we present the results grouped by the compression level and showing the explicit dependence of the output file size on the number of input files.

1. Compression level 3:
   The most important result we observe in Fig. 2.17 is that, just as the execution time grows disproportionally with the number of input files, so does the output file size. Furthermore, it can be seen that there is a small influence of the block size on the output file size: the larger the block size, the smaller the output size. The reason is that the compression algorithm works on the block level and is expected to be able better to compress one block of e.g., 50000 elements than 10 blocks of 5000 elements.

2. Compression level 6:
   The results for this compression level, shown in Fig. 2.18 are very similar to the previous case.

3. Compression level 9:
   The Fig. 2.19 shows that in this case the results are very similar to the previous two cases.

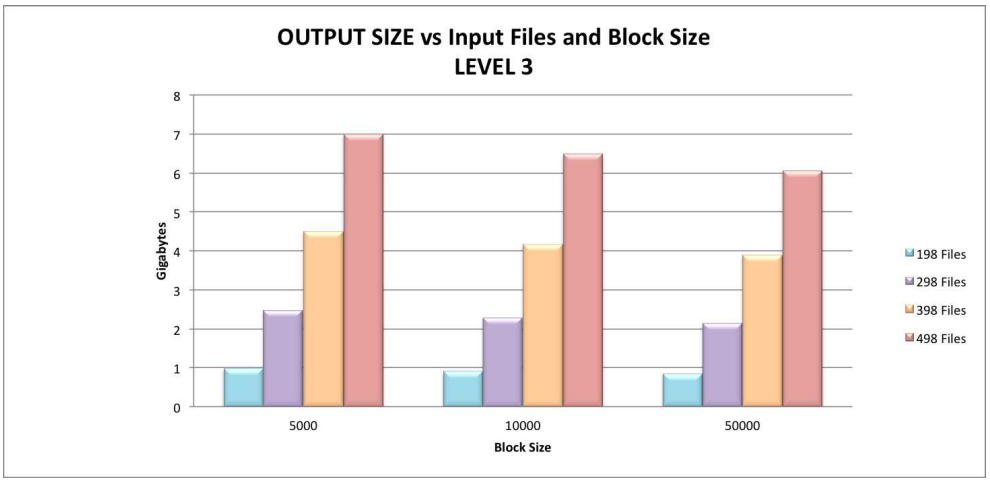

Figure 2.17: Size as a function of block size and number of files for compression level 3.

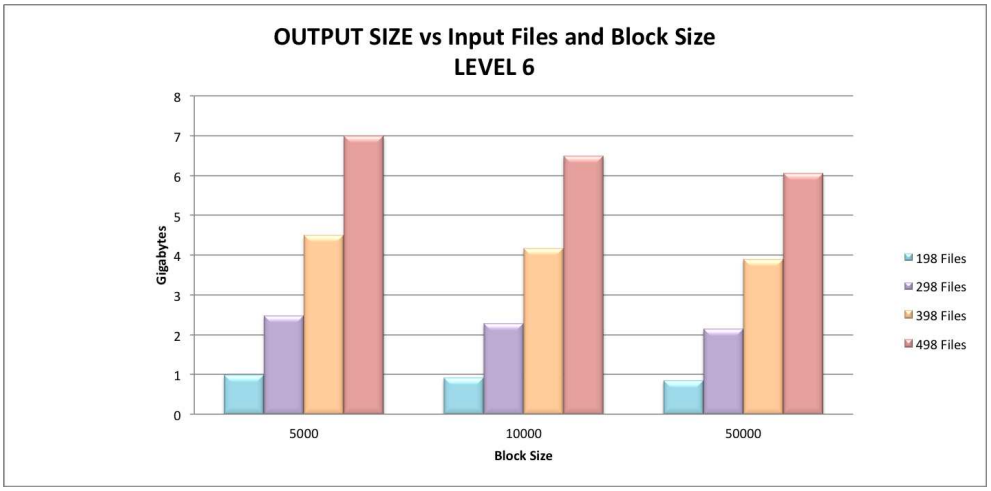

Figure 2.18: Size as a function of block size and number of files for compression level 6.

SUMMARY OF THE PREPROCESSOR TESTS

In summary, we can conclude that the best value of block size is the one that has the smallest execution time and the block size equal to 10000. Regarding the level of compression, tests with level 9 have an execution time which is on average the same as for the lower compression levels. However, they produce a file size which is noticeably smaller than the lower compression levels.

The Fig. 2.20 shows the dependence of the execution time on the block size for all tests. The four groups of lines correspond to the different numbers of input files. For the 498 files we see that the best results are obtained for the block size 10000. However, one has to keep in mind that the in the realistic applications the number of blocks can be rather large for small block sizes

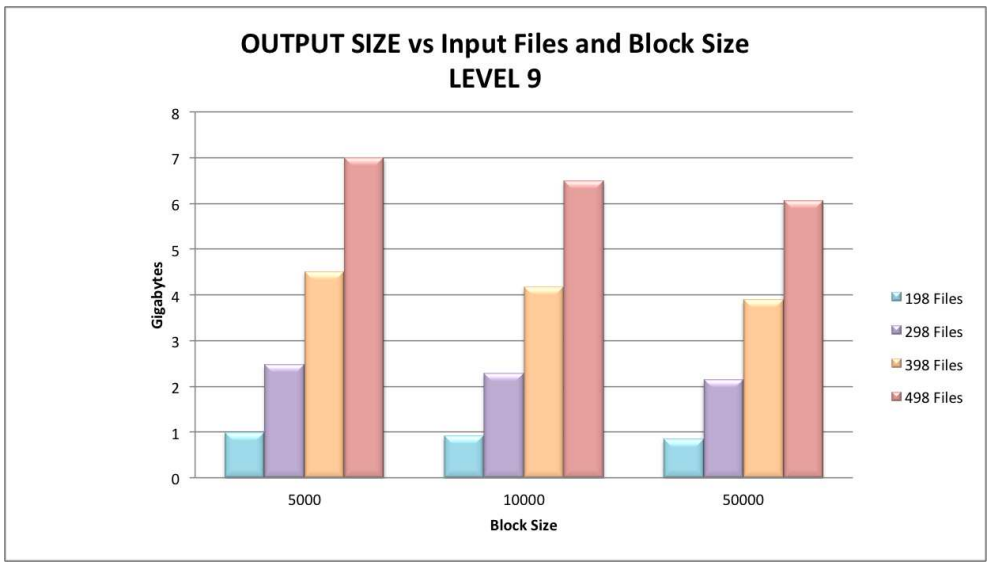

Figure 2.19: Size as a function of block size and number of files for compression level 6.

(for this reason we discard the block size 5000 as not practical).

In Chap. 5 we discuss the structure and the internal organization of the output file.

2.2.3 Testing the postprocessing

As discussed in Sec. 2.1.1, the *SPEV* postprocessor reads the blocks from the preprocessed file and calculates the contributions to the virtual detector. The postprocessor reads the blocks in sequence. Each block has to be read from the HDF5 into memory and uncompressed (if necessary). If the block size is very small the number of blocks is going to be very large, and the time spent reading and uncompressing blocks is going to be prohibitively large compared to the time spent performing actual computations. Therefore, in this section we limit ourselves to block sizes 10000 and 50000. In Fig. 2.21 we show the execution time as a function of the block size for different compression levels. We see that there is a non monotonic decrease in performance as the compression level grow, but the difference in the execution time between the runs with uncompressed and compressed preprocessed files is below 5%. There is no significant difference between different levels of compression within the accuracy level of the time measurement.

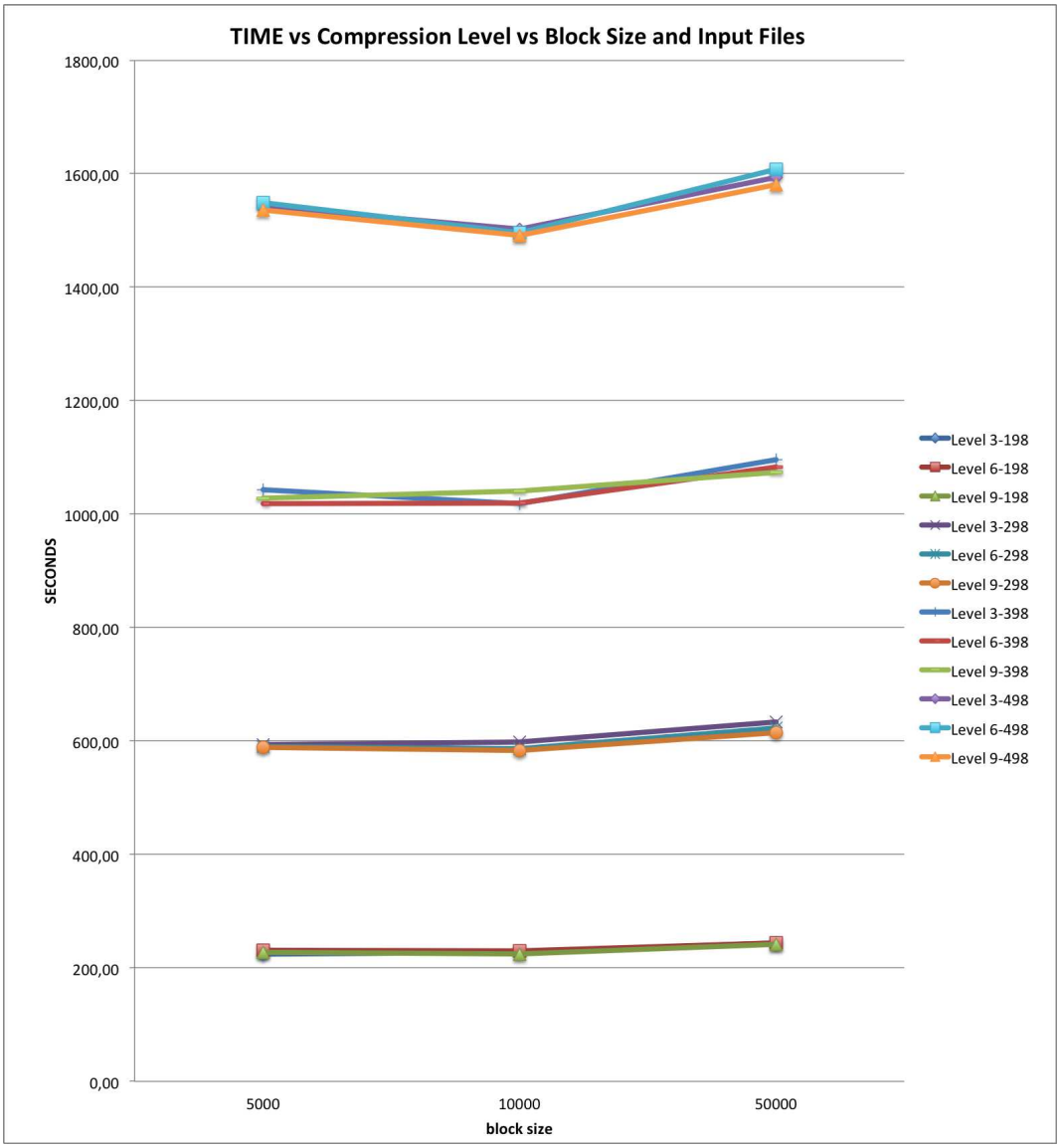

Figure 2.20: Summary of preprocessing tests: execution times as a function of the block size for all tests. In the legend the compression level and the number of input files are indicated. The values of the execution time obtained for different block sizes but a fixed number of input files are joined by solid lines. Since larger number of input files produce larger execution times, the different sets having the same number of input files naturally stack above each other in growing amount of execution time.

2.2.4 Conclusion

The results of the tests of both preprocessor and postprocessor point towards the fact that it makes sense to use the compression when writing preprocessed

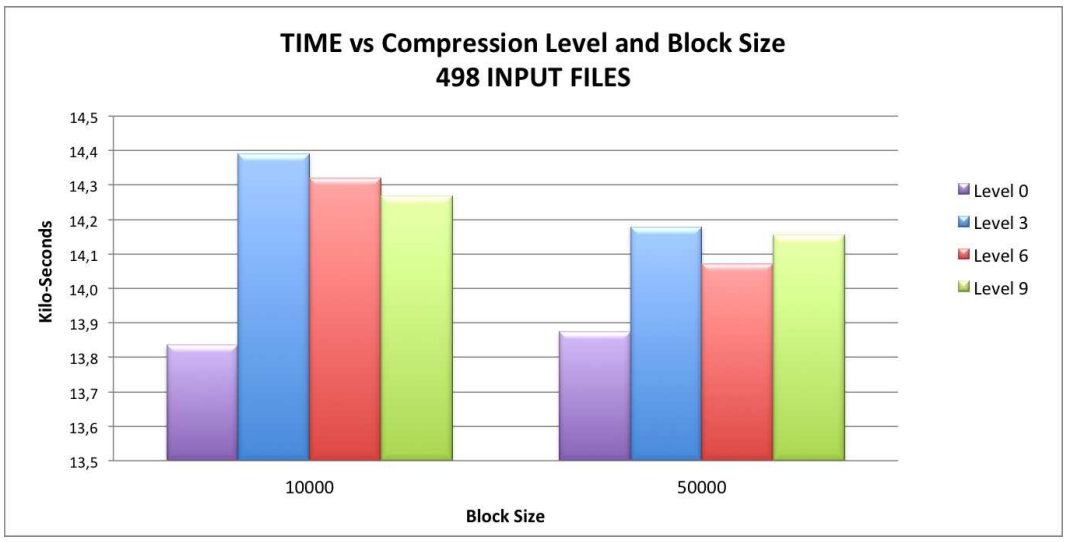

Figure 2.21: Execution time of the postprocessor as a function of the block size and the compression level. The preprocessed files containing blocks were generated by the preprocessor using 498 input files (see Sec. 2.2.2). The times given are average of several repetitions of the same test, measured with the command line `time`. The accuracy is approximately 2% due to the shared usage of the CPU between the program and the processes of the operative system.

files. However, the preprocessor in general has to be executed only once, while the postprocessor will be executed multiple times. Furthermore, the execution time of the postprocessor is, in general, longer than that of the preprocessor. Therefore, we find that the best option for *SPEV* is the block size of 50000 and the compression level 9.

Chapter 3

Optimization of use of hardware resources

In this chapter we discuss the work done in optimizing the use of hardware resources by modifying the codes *MRGENESIS* and *SPEV* to run side-by-side on a multiprocessor machine. Furthermore, the code which produces images from the RMHD simulations was also modified to run in parallel with the other two codes.

3.1 Simultaneous execution of different codes

As was discussed in the previous chapter, *SPEV* needs to preprocess the output files produced by *MRGENESIS* in order to generate an intermediate preprocessed file from which the final images can be computed. Since there is no feedback from *SPEV* to *MRGENESIS*, there is no need for *SPEV* to wait until *MRGENESIS* has finished the whole simulation, and it can start preprocessing the output files as soon as they are produced. The same is true for the *PARPLOT* code, which is a combination of a shell script, C++ code and an Asymptote [1] script which produces high-resolution 2D images of the relativistic fluid. Since *MRGENESIS* is CPU and memory intensive, while *SPEV* preprocessor and *PARPLOT* are I/O intensive, they can run in parallel without causing congestions. The reason is that *SPEV* and *PARPLOT* would only be active for a short time after a new HDF5 file is written by *MRGENESIS*, and would be inactive while waiting for the next one to arrive.

The Fig. 3.1 shows the flow-chart of the algorithm that is executed every time a HDF5 file is written by *MRGENESIS*. After initialization *MRGENESIS* executes a loop which periodically produces an output HDF5 file (see Fig. 3.2 for an schematic flow chart of *MRGENESIS*). The output files

[1] `http://asymptote.sourceforge.net`

are written in parallel by all MPI threads using HDF5 parallel interface. *PARPLOT* and *SPEV* are initially inactive and wait until such output file appears. Once that happens, they read the file and process it. Meanwhile *MRGENESIS*, which does not need any feedback from the other two codes, keeps on calculating and eventually produces the next output file. If *SPEV* and *PARPLOT* have finished processing the previous file early, they are inactive until the new file appears, otherwise they will process all new files in order of appearance. This procedure uses both CPU and I/O resources of the machine very efficiently, since, as mentioned above, *MRGENESIS* uses CPUs at the same time as *PARPLOT* and *SPEV* use the I/O system.

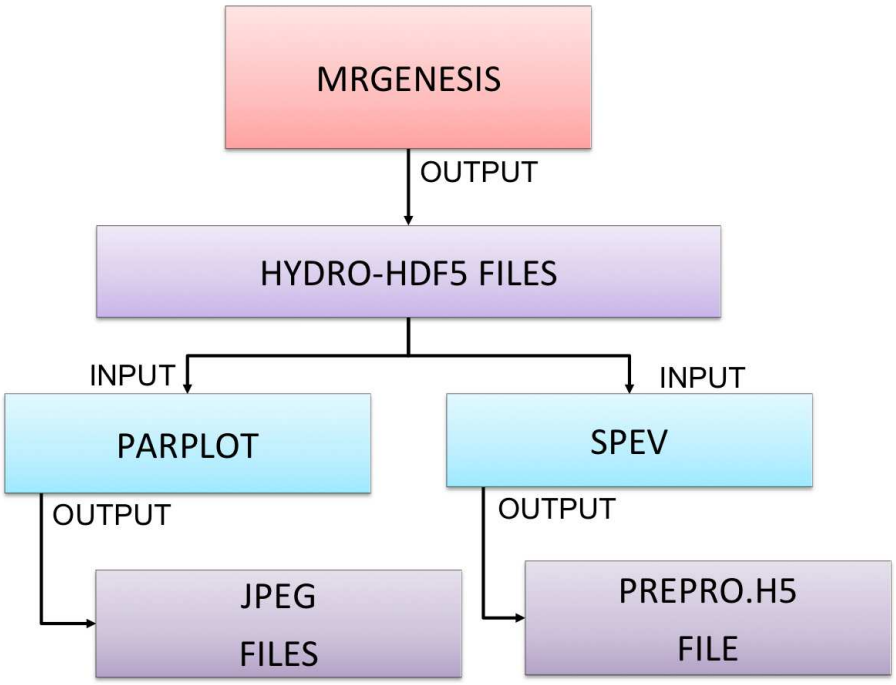

Figure 3.1: Data flow diagram of the simultaneous execution of *MRGENESIS*, *PARPLOT* and *SPEV*. *MRGENESIS* periodically writes an output file which *PARPLOT* and *SPEV* process. *MRGENESIS* does not need to wait for any feedback from *PARPLOT* and *SPEV*.

The figures 3.3 and 3.4 shows the tasks flow before and after the modification of the codes, respectively. We note that the modifications of the code will still leave the possibility of running in the old configuration (Fig. 3.3), for example in case one needs to repeat *SPEV* preprocessing using a different set of parameters after the *MRGENESIS* simulation has already finished.

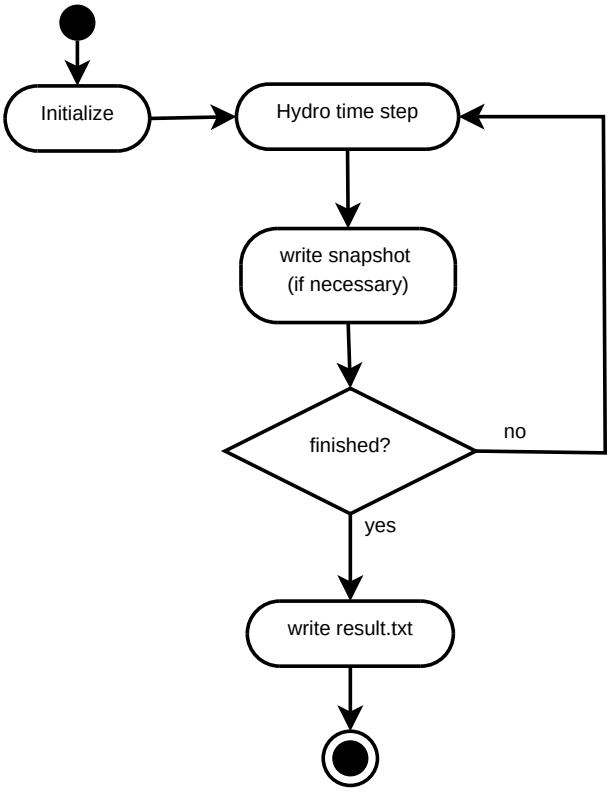

Figure 3.2: Zoom of the data flow diagram of *MRGENESIS*. These steps are implemented inside the main loop. The code is parallelized using MPI and OpenMP.

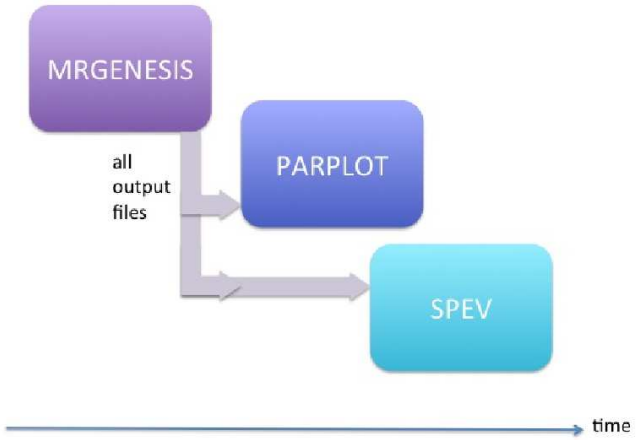

Figure 3.3: Tasks flow before modification: both *PARPLOT* and *SPEV* must wait until *MRGENESIS* has finished creating all output files before they start processing them.

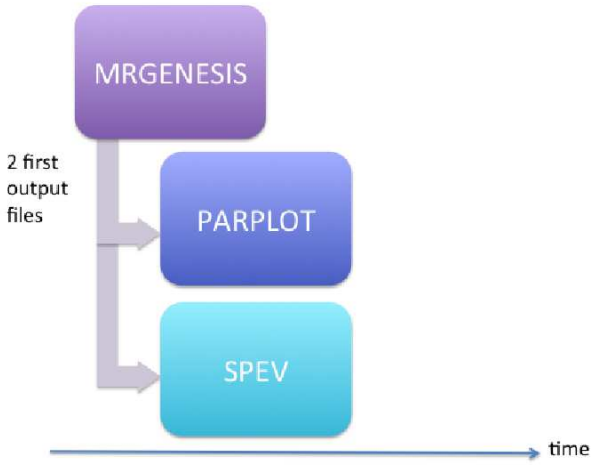

Figure 3.4: Tasks flow after modification: *PARPLOT* and *SPEV* can process output files while *MRGENESIS* is still running.

3.2 Code modifications

In this section we describe how all three codes (*MRGENESIS*, *SPEV* and *PARPLOT*) have been modified to achieve the tasks flow as shown in Fig. 3.4.

3.2.1 Modification of *MRGENESIS*

MRGENESIS is written in Fortran 95 and its algorithm in simplest form can be seen in Fig. 3.2. Although there is no direct impact of what *SPEV* and *PARPLOT* do to the functioning of *MRGENESIS*, it nevertheless had to be slightly modified. The reason is that *MRGENESIS* has to communicate to *SPEV* and *PARPLOT* that it has finished the computation and that the last file has been written to disk. Otherwise, *SPEV* and *PARPLOT* will be stuck in an infinite loop, waiting for an output file which will never appear.

To this end we have modified the source file **main.f**. The flow chart shown in Fig. 3.5 shows the relevant part of the code. Once the computation has been finished the file `result.txt` is written to disk. In case of an execution in parallel environment it is ensured that only the master (rank 0) MPI task will write the file. The source code which is inserted into **main.f** is as follows:

```
call rrhdf5_record(.true.)
if (myrank .eq. 0) then
   open(1, FILE=TRIM(outRoot)//"-result.txt", form='formatted')
   write(1, *) R_file
   close(1)
end if
```

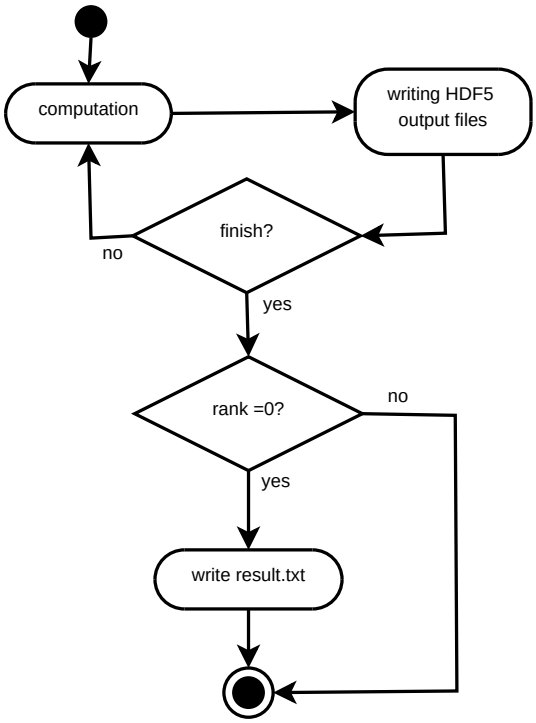

Figure 3.5: Flow chart of the modifications to *MRGENESIS*.

3.2.2 Modifications of *SPEV* (preprocessing)

The preprocessing part of the *SPEV* code reads the *MRGENESIS* output files and injects and evolves the non-thermal particles based on the information in those files. For technical reasons *SPEV* preprocessor always needs to have at disposal the current and the next file. Therefore, for *SPEV* to preprocess the first file it needs to wait at least until the second file has been output as well. For the same reason, the last file that *SPEV* will preprocess is the penultimate *MRGENESIS* output file, though it will use some information from the last *MRGENESIS* output file as well.

Taking this into account the modifications need to be applied in two different source code files.

MODIFICATIONS OF HYDROREADER.F

The file **hydroreader.F** contains the subroutines which read the HDF5 files output by *MRGENESIS*. What had to be included was a code which will check that the current and the next file exist before they can be read and

passed to the main preprocessor code (see next subsection). Fig. 3.6 shows the flow chart of the modified version of hydroreader.F.

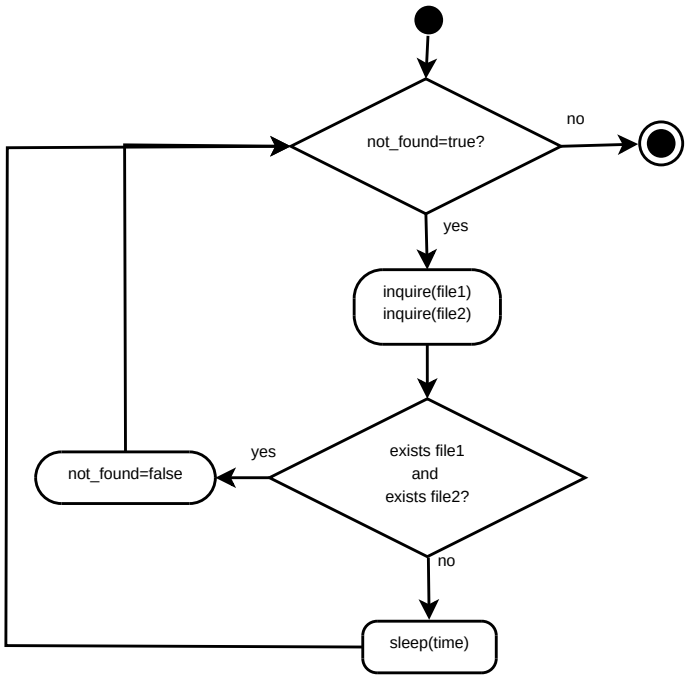

Figure 3.6: Flow chart of the modified algorithm in the hydroreader.F. The existence of the current and the next file (file1 and file2, respectively) is probed. As long as neither or only one exist, the routine sleeps for a short interval of time and checks again.

The following is a source code of the modification. *MRGENESIS* output files are numbered and the code inquires whether two consequent files exist and only proceeds once both are found on the disk.

```
      WRITE(file_ext, "(i5.5)") number
      file_name1 = TRIM(root)//'-'//TRIM(file_ext)//'.h5'
      WRITE(file_ext, "(i5.5)") number + 1
      file_name2 = TRIM(root)//'-'//TRIM(file_ext)//'.h5'

!     Check if the files exist, if not then wait for a while and check again

      DO WHILE (notfound)
         INQUIRE(FILE=file_name1, EXIST=exist1)
         INQUIRE(FILE=file_name2, EXIST=exist2)

         IF (exist1 .AND. exist2) THEN
            notfound = .false.
         ELSE
            CALL sleep(time)
         ENDIF
      ENDDO
```

MODIFICATIONS OF SHPRSPEV2.F

The file **shprspev2.F** contains the main loop of the *SPEV* preprocessor. Considering the modifications of *MRGENESIS* (Sec. 3.2.1) the modified version of shprspev2.F has to able to handle the following two cases:

- **case 1:** the simulation has finished before *SPEV* preprocessing starts; or
- **case 2:** the simulation finishes when *SPEV* preprocessor is already running.

Obviously, if the simulation has finished before the *SPEV* preprocessor starts there is no need to verify the existence of files since they should already all be in place. To handle both possibilities, we use the `result.txt` control file written by *MRGENESIS* (Sec. 3.2.1). As long as that file does not exist *SPEV* assumes that *MRGENESIS* is still running. Once `result.txt` appears, it is assumed to contain the number of the last file written by *MRGENESIS*.

The user of *SPEV* preprocessor supplies the first and last file (integer variables $start\_file$ and end_file) she/he wants to preprocess. If $end\_file \geq start\_file$ it is assumed that *MRGENESIS* has already produced all files (case 1). If $end\_file == -1$ then it is assumed that *MRGENESIS* is still running (case 2). Fig. 3.7 shows the flow chart of the algorithm which implements this functionality.

The source code in Fortran is as follows:

```
      cur_file = start_file
      notend = .true.
      file_name = TRIM(input_root)//"-result.txt"

      if (end_file .eq. -1) then
         INQUIRE(FILE=file_name, EXIST=exist)

         if (exist) then
            open(1, FILE=file_name, form='formatted')
            read (1, *) end_file
            end_file = end_file - 1
            close(1)

            if (cur_file .eq. end_file) then
               notend = .false.
            endif
          else
             end_file = cur_file + 1
         endif
      endif

!     loop over all hydro files
```

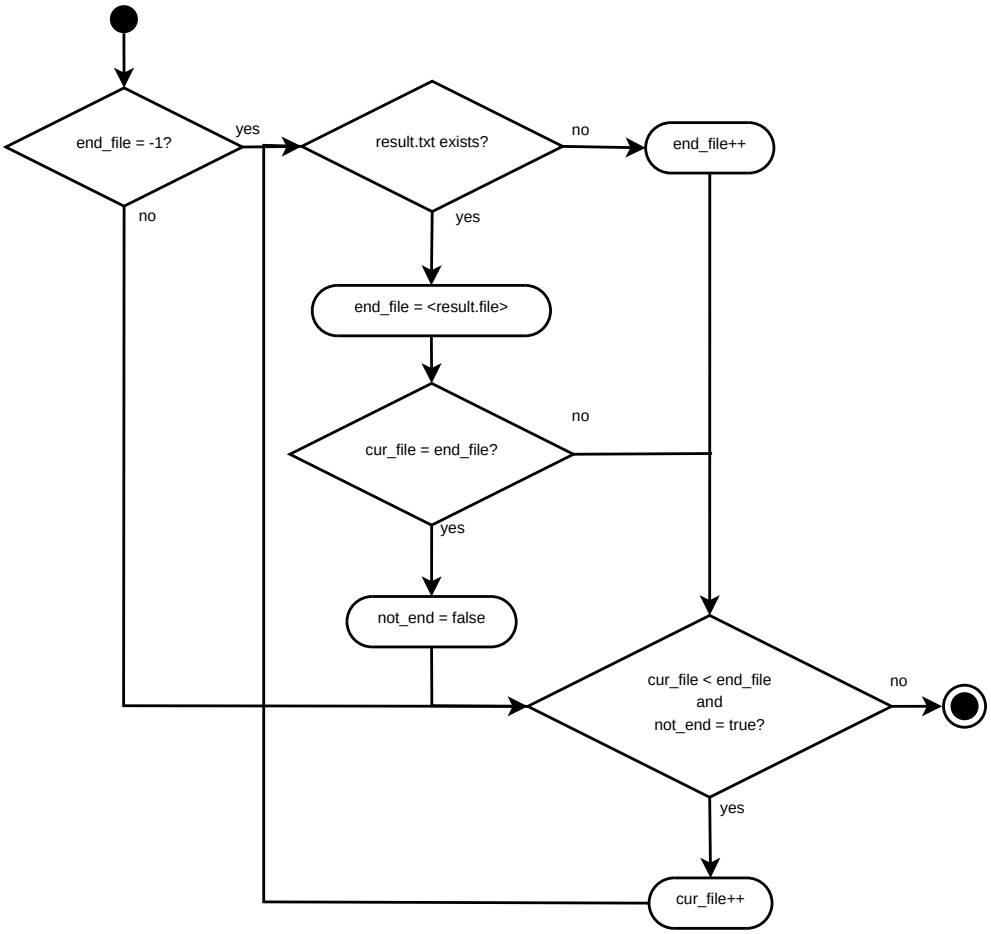

Figure 3.7: Flow chart of the modifications of shprspev2.F. If end_file is not -1, then case 1 is assumed and *SPEV* preprocesses all files from $start\_file$ to end_file. Otherwise, case 2 is assumed and end_file is incremented every time a new output file appears until such time when `result.txt` appears, which signifies that *MRGENESIS* has produced its last file. After that point the procedure is identical to case 1 for the remaining unprocessed files.

```
DO WHILE (cur_file .lt. end_file .and. notend)

    (...)

    cur_file = cur_file + 1

    INQUIRE(FILE=file_name, EXIST=exist)

    if (exist) then
       open(1, FILE=file_name, form='formatted')
       read (1, *) end_file
       end_file = end_file - 1
```

```
            close(1)
            if (cur_file .eq. end_file) then
               notend = .false.
            endif
         else
            end_file = end_file + 1
         endif
      ENDDO
```

3.2.3 Modifications of *PARPLOT*

The script *PARPLOT* processes *MRGENESIS* output files and generates images in JPEG format which are used to visualize the state of the simulation or to generate movies. The algorithm is analogous to the *SPEV* preprocessor and the flow charts shown in Figures 3.6 and 3.7 are valid for *PARPLOT* as well. We have to point that this process is done in parallel by shell threads and we have modified the script in order to allow one thread to continue the computation when new files are written by *MRGENESIS* while the rest of the threads are waiting their turn sleeping.

This is the source code of the shell script modification: **parplot.sh:**

```
(...)
curfile=$startfile
notend=false
notadpt=true
filename=$iprefix"-result.txt"
finished=false

#Check if file with number of input files exist
if [[ $endfile -eq -1 ]];
then
    notadapt=false
    if [ -s $filename ];
    then
        endfile=‘cat $filename‘
        endfile=‘expr $endfile \- 1‘
        echo "Found endfile=", $endfile
        notend=false
    else
        endfile=‘expr $curfile \+ 1‘
        notend=true
    fi
fi

# process all files but last in parallel
while [[ "$curfile" -lt "$endfile" ]] || [ "$notend" == "true" ];
do
    fname=$iprefix-‘printf "%05d" $curfile‘.h5

    while ! [ -s $fname ] && [ "$finished" == "false" ];
```

```
    do
        if [ "$notadapt" == "false" ] && [ "$notend" == "true" ];
        then
            if [ -s $filename ];
            then
                endfile=‘cat $filename‘
                endfile=‘expr $endfile \- 1‘
                notend=false

                if [[ "$curfile" -ge "$endfile" ]];
                then
                    finished=true
                fi
            fi
        fi
        sleep 1;
    done

    if [ "$notadapt" == "false" ] && [ "$notend" == "true" ];
    then
        sleep 1;
    fi

    if [ "$finished" == "false" ];
    then
        proc1file $paramsf $iprefix $oprefix $curfile $minrad $trtr
$totpix $numbins $var1 $var2 $var3 $var4 &
        curfile=‘expr $curfile \+ 1‘
# Avoid execution of more than nproc processes at the same time
        nrwait $nproc
    fi

    if [ "$notadapt" == "false" ] && [ "$notend" == "true" ];
    then
        if [ -s $filename ];
        then
            endfile=‘cat $filename‘
            endfile=‘expr $endfile \- 1‘
            notend=false
        else
            endfile=‘expr $endfile \+ 1‘
        fi
    fi
done
(...)
```

3.3 Performance test

We have performed several tests to demonstrate the reduction of the total time execution of the three parts of the process. The images obtained as result of the tests performed are illustrated in Fig. 3.8, where on the left panel we can see the test with low resolution (1500 radial by 250 angular numerical zones) and on the right pannel the same test with high resolution results (6000 radial by 500 angular zones). We have compared both modes of execution: the serialized execution[2] (using unmodified codes) and the concurrent execution with the modifications implemented.

The first test is a low-resolution (1500×250 numerical cells) hydrodynamic simulation of a relativistic jet propagating into the external medium. This is a relatively short execution with 48 processors in total and the results are as follows:

- **Serialized execution:**

MRGENESIS 30 CPUs MPI	*SPEV* 12 CPUs OPENMP	*PARPLOT* 6 CPUs SHELL MULTITHREADING	TOTAL TIME
339 s	692 s	948 s	1979 s

- **Concurrent execution:**

MRGENESIS 30 CPUs MPI	*SPEV* 12 CPUs OPENMP	*PARPLOT* 6 CPUs SHELL MULTITHREADING	MAX TIME
336 s	701 s	954 s	954 s

The serialized run needs a total time of 1979 seconds, since in this mode we need to sum the execution time of the three stages. In the concurrent mode we have launched the three parts at the same time. The execution of the three codes overlaps, and *PARPLOT SPEV* waits until the output files of *MRGENESIS* are produced. When the output is written, both processes start their computations. Since this is a low-resolution run, *MRGENESIS* produces files much faster than either *SPEV* or *PARPLOT* can process them. The last process to finish is *PARPLOT*, for this reason the total time of the concurrent process is the same of total time of the execution. With these results we obtain a speedup of 2.97: $SPEEDUP$: 1979 seconds /954 seconds = 2.07

[2] "Serialized" refers to the fact that we execute one code after another. The codes themselves are still executed in parallel.

The second test is a high-resolution version of the same run (6000×500 numerical cells).

- **Serialized execution:**

MRGENESIS 30 CPUs MPI	*SPEV* 12 CPUs OPENMP	*PARPLOT* 6 CPUs SHELL MULTITHREADING	TOTAL TIME
16554 s	1008 s	1094 s	18656 s

- **Concurrent execution:**

MRGENESIS 30 CPUs MPI	*SPEV* 12 CPUs OPENMP	*PARPLOT* 6 CPUs SHELL MULTITHREADING	MAX TIME
start 09:44:59 finish 14:15:34 16235 s	start 09:44:59 finish 14:15:39 16238 s	start 09:44:59 finish 14:15:46 16247 s	start 09:44:59 finish 14:15:46 16247 s

In the this test, the serialized execution needs a total time of 18656 seconds. The concurrent execution, that overlaps the three processes, starts at 09:44:59. The last process to finish is *PARPLOT* at 14:15:46, with total time execution of 16247 seconds. With these results we obtain a speedup of 1.15: $SPEEDUP$: 18656 seconds /16247 seconds = 1.15. **The time saved with the concurrent execution are 40 minutes.**

3.3.1 Conclusions

The speedup obtained for the more accurate (higher resolution) run is poorer than that of the less accurate one. The main reason is that the total execution time is dominated by the calculations carried by *MRGENESIS*, being both *SPEV* and *PARPLOT* a small correction to the overall execution time. Obviously, in this case, then number of processors allocated for the *MRGENESIS* execution is too small. Employing a larger number of CPUs for the most time consuming code shall be a rather obvious way to improve the speedup of the concurrent execution. However, during this thesis, we did not have time to find the optimal number of processors that one must use to execute each code concurrently.

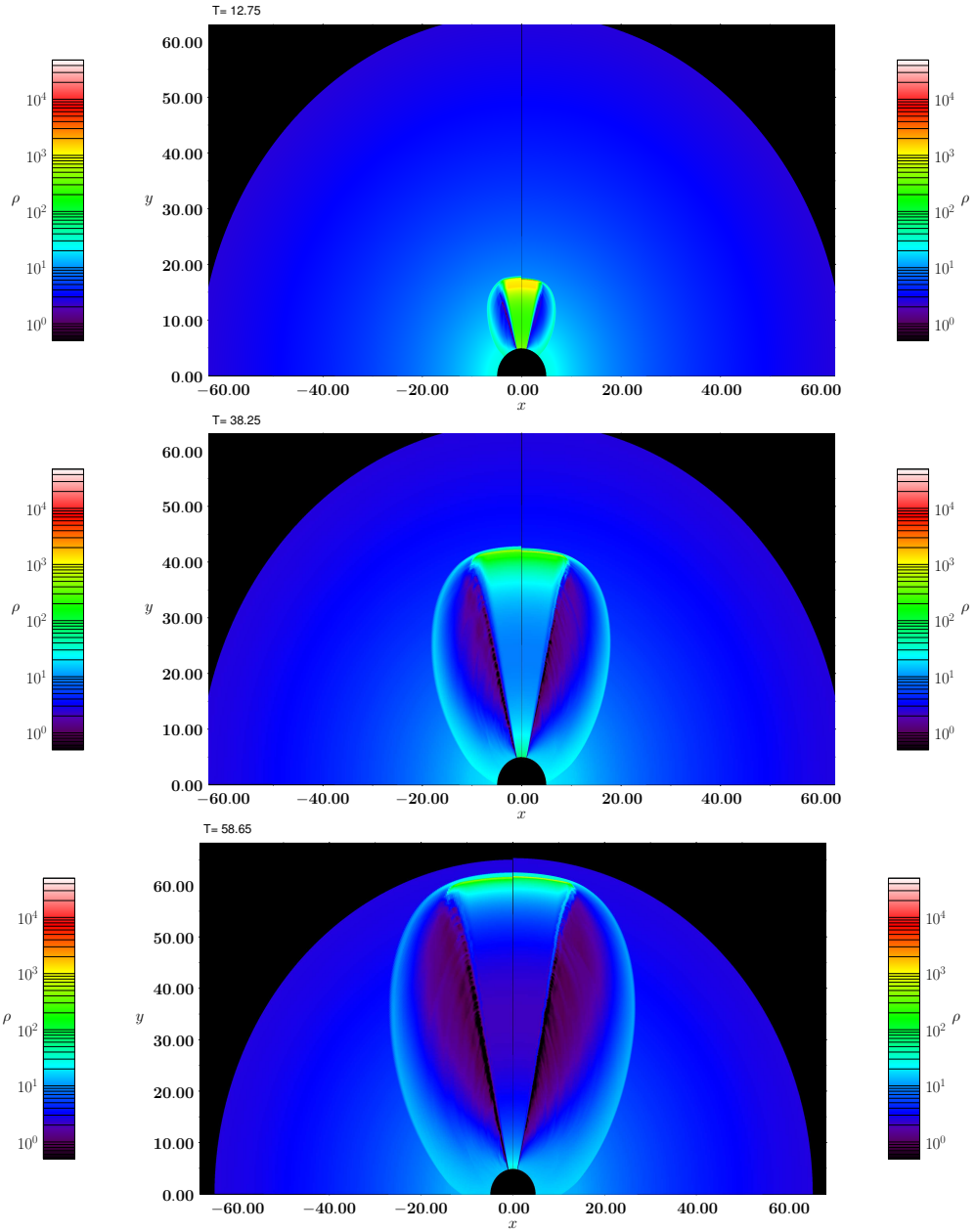

Figure 3.8: Images produced by *PARPLOT* for the tests performed in Sec. 3.3. We plot ρ, the number of the fluid particles (in units of particles per cm^3) at three different times. The simulations have been run in spherical coordinates assuming axial symmetry. The left panel shows the results of the low resolution (1500 radial by 250 angular numerical zones) simulation, while on the right we display the high resolution results (6000 radial by 500 angular zones). We see a relativistic jet (initially green) expanding into a stratified external medium. It slows down and gradually becomes less dense. The observed emission is produced at the shock wave (discontinuity at the head of the jet). It can be seen that the higher resolution run exhibits sharper features than the low resolution one.

Chapter 4

Optimization of memory management

As was discussed in Sec. 2.1, and as can be seen in Fig. 2.1, *SPEV* postprocessor uses significant amounts of RAM during its execution. Furthermore, memory space is allocated dynamically, grows during the execution and the total required memory capacity *cannot be calculated in advance*. This makes the *SPEV* memory management rather challenging. In Sec. 4.1 the *SPEV* virtual detector is explained and its demanding memory requirements are explained. Sec. 4.2 details the improvement of the memory management via linked lists. Finally, in Sec. 4.2.3 we verify that the results produced by the improved code are identical to those produced by the old and demonstrate the improvement of efficiency of memory management.

4.1 *SPEV* virtual detector and its memory requirements

In this section we give a brief description of the *SPEV* virtual images/cap02-postproces.epsdetector and why it is important to improve its memory management.

4.1.1 Radiative transfer in *SPEV*

As discussed in Sec. 2.1.1, in the *SPEV* postprocessor there is a *virtual detector*, which simulates a telescope on the Earth or a detector on board of a satellite. The purpose of *SPEV* is to correctly compute the photon flux through each of the detector pixels.

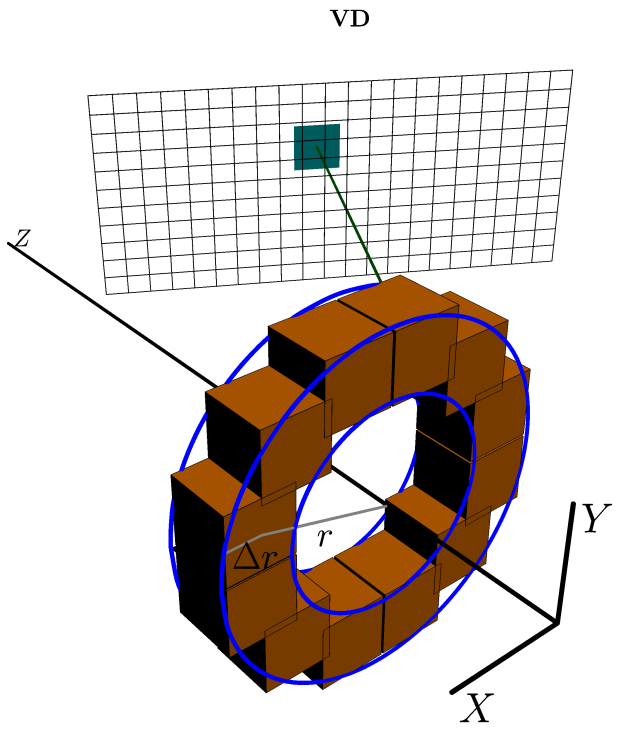

Figure 4.1: Illustration of the *SPEV* virtual detector (VD in the image): a collection of brown cubes represent non-thermal particles in a jet. Each particle is observed by one or more pixels in the virtual detector. The observability of particles depend on the orientation between the normal to the virtual detector and the jet axis (Z), as well as the time and position of the particles.

The Fig. 4.1 shows the typical configuration during the execution of the *SPEV* postprocessor. A collection of particles (brown cubes) is emitting radiation which can be observed in the pixels of the virtual detector. Depending on the size and orientation, each particle can be observed in one or more pixels (which pixels observe one particle are indicated by the dark rectangle). The purpose of the postprocessor is to collect all the contributions from different particles to each pixel of the virtual detector by reading and processing the data stored in the blocks of the preprocessed file (see Sec. 2.2 for an analysis of the preprocessor). Once all blocks in the preprocessed file have been read, there can be no new contributions to the pixels, and the value of the radiative flux in each pixel is determined. The flux is computed by solving the radiative transfer equation for each pixel separately. This involves sorting all contributions (on the *line of sight*) according to the distance from the pixel, and then accumulating the flux starting from the most

distant contribution and moving towards the detector, taking into account the emission of new radiation and the absorption of the existing radiation by each element of the line of sight. The number of contributions (elements) strongly depends on the type of the problem which is being solved, as well as the orientation of the jet and the properties of the emitting particles. If there are no contributions to a pixel at all, the number of elements to a line of sight is zero. Where there is a lot of emission, the number of elements can be quite high (see Sec. 4.2.3).

4.1.2 Virtual detector memory management

As discussed in the previous section, the key task of the postprocessor is the storage of all contributions (lines of sight) to each virtual detector pixel while the preprocessed file is being read and the radiation from non-thermal particles computed. Since all lines of sight start empty, this means that the memory which *SPEV* uses *grows* during its execution. In the current implementation this was accomplished by allocating for each pixel an initial line of sight with *lres* elements, where *lres* is a parameter supplied by the user. For each pixel there are two counters: *alloc_los*, which keeps track of the total memory available, and *num_los*, which counts the actual number of elements in a line of sight. As new elements are added to a given line of sight, *num_los* grows until it becomes equal to *alloc_los*. At this point, a copy of the line of sight is made, the old line of sight is deallocated and allocated such that the new size is *fac* times of the old size, where the constant $fac > 1$. Afterward the data is copied back into the newly allocated line of sight. Typical values of fac are between 1.2 and 1.5.

Obviously, this is a very inefficient process, both from the execution time and the memory usage point of view. Because the allocated memory grows exponentially every time a line of sight needs to be extended, and because there is no guarantee that by the end of the execution *all* available elements are actually going to be used, the inefficiency can be substantial (see Sec. 4.2.3). Therefore, we have decided to implement the lines of sight as linked lists, in order to reduce this particular inefficiency.

4.2 Implementation of linked lists in *SPEV* virtual detector

In this section we give details about the implementation of the linked lists in the virtual detector. In Sec. 4.2.1 we compare the old and the new algorithm,

while in Sec. 4.2.2 we give the details of the implementation of the new code. Finally, in Sec. 4.2.3 we test the new code and compare its performance with that of the old implementation.

4.2.1 Comparison of old and new algorithms

As discussed in Sec. 4.1.2, the previous implementation needed to allocate more memory than it finally used. The data flow shown in Fig. 4.2 illustrates the parallel process of checking whether the line of sight (array *los_data*) is full, and the subsequent reallocation of more memory. As can be seen, there is a number of memory copying operations and temporary arrays, and the allocated space grows exponential during the code execution. In the boxes grey color denotes the space occupied by line of sight elements, while the white portion shows the unused portion. There is a lot of waste, both of memory and of CPU time.

The improvement was in the implementation of linked lists (Fig. 4.3). The nodes of the lists are created dynamically in parallel when the last is full for the current thread. All of them have the same size (*lres*). Now the reserved memory does not grow exponentially, but linearly. The only waste of memory occurs is in the very last node, whose size is small compared to the total number of elements in the list. In Sec. 4.2.3 we study which is the best value for the node size.

The Fig. 4.4 shows the data structure of the new memory management model. Each pixel of the virtual detector (a three-dimensional structure) has a linked list representing its line of sight associated to it (*los_data*). Technically this is achieved by associating three separate linked lists to each pixel: *list_dist*, containing the distance of the line of sight element, *list_emiss* containing the emission and *list_absor* containing the absorption in the element.

4.2.2 Implementation

The implementation of the linked lists is written in a Fortran module. The module contains all the routines needed to implement a fully functional linked list in the postprocessing part of the *SPEV* code. We have implemented auxiliary routines which determine list sizes, initialize pointers, get and set elements, append nodes and destroy lists.

In addition to these general routines we have added routines which are specific to the *SPEV* postprocessor algorithm, such as a routine which copies all the data from the linked list into a vector. This is useful before performing the radiative transfer (Sec. 4.1), since the elements are only going to be read once and in order. Using a more general routine for getting each element of

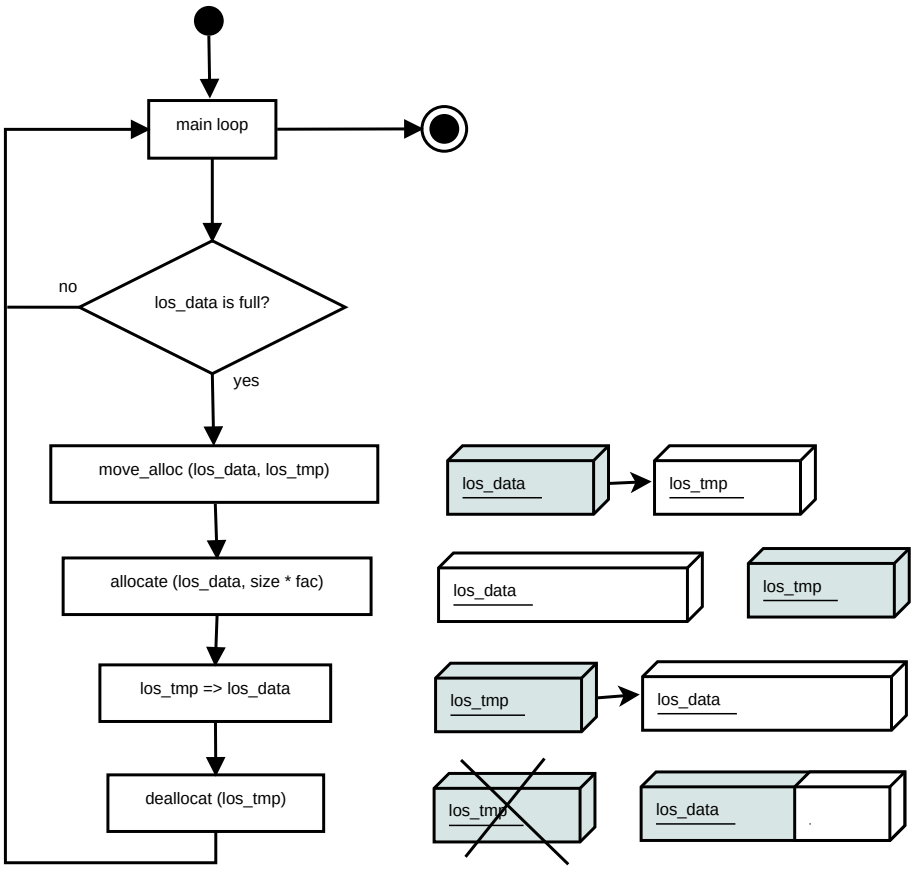

Figure 4.2: Data flow of the old memory management. Shown are the steps of the main *SPEV* loop, which is parallelized using OpenMP. In this main loop the line of sight elements are added to the *los_data* array. There is a conditional jump which determines when *los_data* has been filled. If that is the case, the current *los_data* is moved to a temporary array, then it is reallocated with a larger capacity, and then the data are copied from the temporary array *los_tmp* to *los_data*. Afterward the temporary array is destroyed. As can be seen, each reallocation involves two memory copies and two allocations of arrays, creating a performance overhead.

the list separately would result in unnecessary overheads, since each access has to point to the head of the list and move node by node.

The source code of the *list_losdata* module can be found in Appendix 1.1.

4.2.3 Tests

In order to validate the performance of the new code, we need to verify that the results are identical to that of the new code, and also that the memory consumption has indeed decreased with respect to that of the old code.

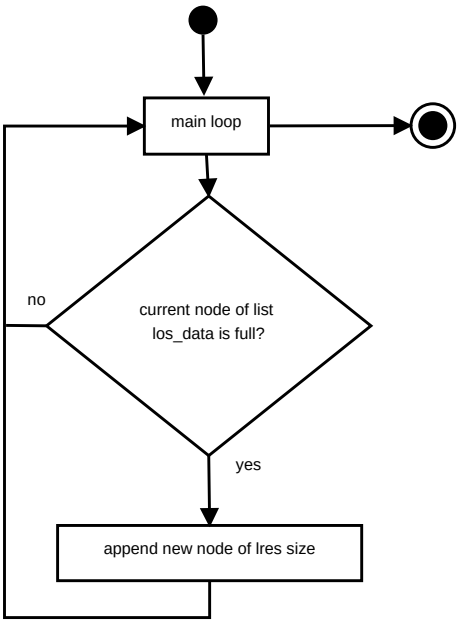

Figure 4.3: Data flow of the new memory management. In the main loop (OpenMP parallelized) the line of sight elements are added to the most recently created node of the linked list. There is a conditional jump to determine whether the node is full. If that is the case, a new node is added. As a difference to the old implementation (Fig. 4.2), there is no need to use temporary arrays, resulting in savings both in performance and in memory usage.

VERIFICATION OF RESULTS

To verify the results we use an output file proceed by the *SPEV* preprocessor on the results of a simulation of a relativistic jet expanding into an external medium. We use the same preprocessed file for both old and the new code, and compute the radio light curve at five different frequencies (Fig. 4.5).

We run both codes without optimization and without parallelism. Every code produces two output files, one with a suffix "-thick" and another with a suffix "-thin". We verify that both output files produced by both codes are identical:

```
$ h5diff oldout-thick.h5 newout-thick.h5
$ h5diff oldout-thin.h5 newout-thin.h5
```

Furthermore in Fig. 4.5 we verify graphically that the results are identical. We plot the flux (in milli-Jansky) that would be observed by a radio telescope on Earth at different epochs (days since the start of the emission). We use the "-thick" output files. The symbols show the results produced by the old code, while the lines show the results of the new code. We can see that the agreement is perfect, since the lines (new code) go through the center of the

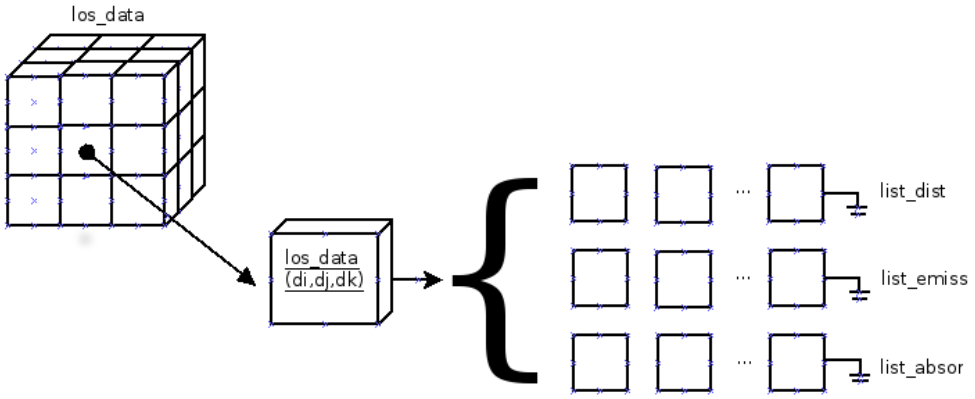

Figure 4.4: Data structure of the new memory management implementation using linked lists. *los_data* is a three dimensional matrix. Each element is a data structure that contains three linked lists: *list_dist*, *list_emiss* and *list_absor*, and other important information such as lists size, number of blocks and the block size. The three lists store the distance, emission and absorption in a line of sight element.

symbols (old code).

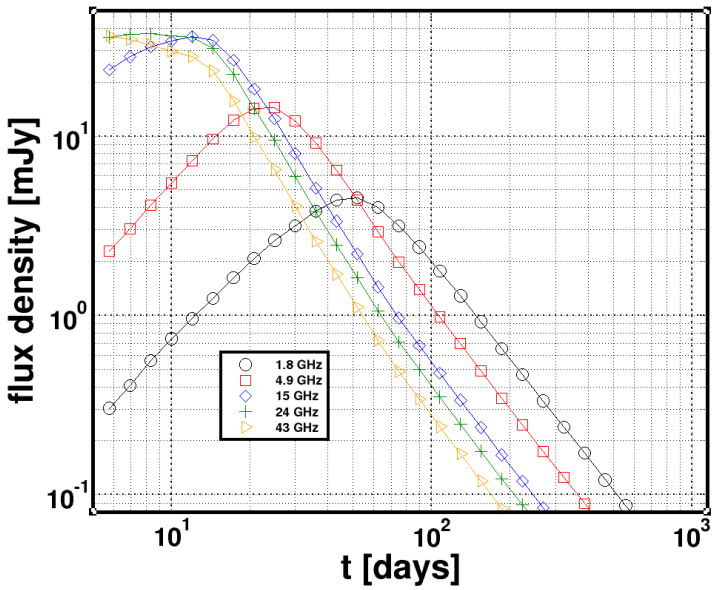

Figure 4.5: Multifrequency radio light curves produced by the old code (symbols) and the new code (lines). Shown is the flux from an evolving jet that would be observed by a radio telescope at different frequencies (given in the legend). As we can see, lines go through the centers of the symbols, which means that the results computed by the old and the new code are identical.

TEST OF THE MEMORY MANAGEMENT EFFICIENCY

As discussed above, the most important parameter of the memory management is *lres*. In the old implementation it determines the initial memory allocated to each pixel. In the new implementation it is the size of the memory allocated to each node of the list.

First, we run a series of tests varying *lres* from 10 to 500. The source code has been compiled using GNU Fortran Compiler using optimization and OpenMP parallelism. The input parameters for *SPEV* postprocessor are:

```
xres =          3001
xmin =    0.0000000000000000
xmax =    300.00000000000000
yres =             1
ymin =    0.0000000000000000
ymax =   1.0000000000000005E-004
tres =            30
tmin =    500000.00000000000
tmax =    100000000.00000000
fres =             1
fmin =    100000000000.00000
fmax =    100000000000.00000
empty_frac =   0.10000000000000001
dens_norm =   1.67262158000000014E-024
length_norm =    10000000000000000.
labToffset =    5.0251890760000002
epsilon_e =   0.10000000000000001
epsilon_B =   5.00000000000000010E-003
pind =    2.2999999999999998
num_bins =            32
trtr =    1.0000000000000000
prtr =   0.50000000000000000
theta_obs =    0.0000000000000000
degrade =             1
redshift =   0.35399999999999998
d_L =   5.56700000000000055E+027
```

The comparison between the memory usage efficiency (ratio of the used to the allocated memory) of the old and new code are given in the following table:

LRES param	OLD code Efficiency	NEW code Efficiency
10	90.301%	99.829%
20	89.363%	99.683%
30	88.386%	99.454%
50	86.667%	99.072%
100	82.565%	98.150%
500	59.779%	91.240%

With these results we conclude that, for tests with low *lres* the efficiency of the new memory management of is more than a 10% higher compared to old code, and is always above 99%. For larger values of *lres* the efficiency of both codes decreases, though it is still above 90% for the new code. The reason for the decrease is that there is now more unused memory in the most recent node compared to the total allocated memory than is the case for small *lres*.

The next parameter which determines the memory usage is the resolution of the virtual detector (*xres*, *yres* and *tres*) and the number of observing frequencies (*fres*). In the test problem we use a one-dimensional virtual detector (*yres*=1) and fix *tres* and *fres*, leaving *xres* free. While the test with *xres*=3001 fit into a memory of a typical iMac, for a test with *xres*=10001 we had to run at a machine with more memory [1]. The input parameters are the same except for that `xres = 10001`.

The following table shows the memory usage efficiency comparison:

LRES param	OLD code Efficiency	NEW code Efficiency
10	90.311%	99.829%
20	89.364%	99.640%
50	86.667%	99.072%
100	82.566%	98.147%
500	59.784%	91.229%

These results are very similar to the ones for *xres*=3000. The memory usage efficiency is always higher using linked lists. We observe that in the worst case, for *lres*=500, the efficiency of the new code is a 91% and the difference of efficiency between both versions is 31,5%.

We have performed an even larger test on *Lluis Vives* cluster [2] to check the efficiency in the more costly and more realistic cases. We have increased again the value of *xres* to 50001. We have recompiled the source code using Intel Fortran Compiler using optimizations and OpenMP. The input parameters are the same except for that `xres = 50001`.

The comparison of the memory usage efficiency is:

LRES param	OLD code Efficiency	NEW code Efficiency
100	82.572%	98.148%
500	59.239%	91.239%

[1] 2 Quad-Core Intel Xeon at 2,93 GHz Processor "Nehalem" with 48GB of RAM

[2] Cluster located at Servei d'Informatica of the University of Valencia. Altix Ultraviolet 1000 with 30 CPU Xeon 7500 hexacore at 2,67GHz, 18MB L3 cache memory on-die, 960 GB of RAM. `http://www.uv.es/lluisvives`

The results in this cluster confirm that the efficiency is always higher in the version with linked lists. It is due to the fact that we always append nodes of the same size, while in the old code the allocated memory grows exponentially.

To see this, consider that the total number of nodes for *list_emiss* and *list_absor* is equal $lres \times fres$ (for *list_dist* it is equal $lres \times 2$). In the worst cases the total unused memory in the last node is $(lres - 1) \times (2 \times lres + 2)$ for every virtual detector pixel. In almost all cases this is much better than was the case in the old code.

In the old code, every time *los_data* was filled, its space was multiplied by a factor of (typically) 1.5. Mathematically the size of each virtual detector pixel at the end of the main loop is $lres \times (2 \times fres + 2) \times (1.5)^n$ where n is the number of memory reallocations for a given pixel. Thus, the allocated memory grows exponentially, and in the last reallocation the size of every element of *los_data* is probably much larger than the original size. In the worst case, when the main loop ends after reallocating the array, the unused space is $(lres \times (1.5)^n) - (lres \times (1.5)^{(n-1)} + 1) \times (2 \times fres + 2)$. This is typically much larger than the memory lost when using linked lists. We have illustrated this behavior in Fig. 4.6

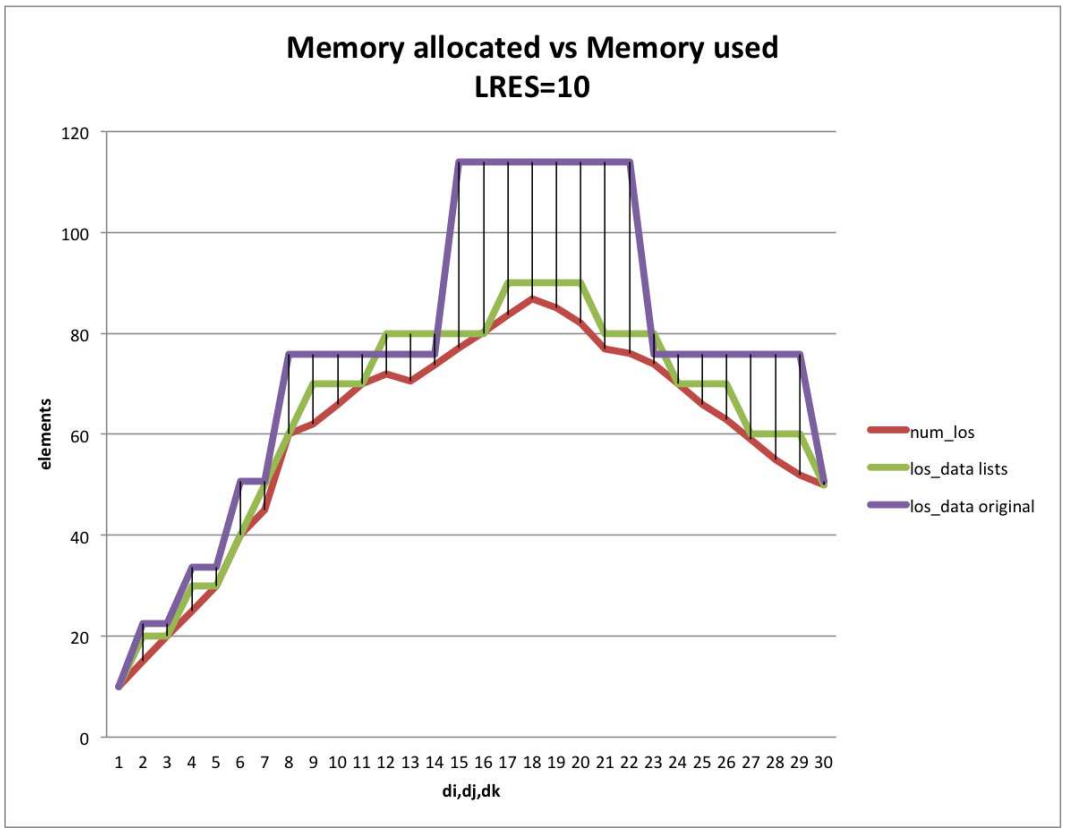

Figure 4.6: An illustration of the memory usage in the old and the new *SPEV* implementation. On the horizontal axis the detector pixel number is plotted (for clarity we only show first 30). On the vertical axis we show the total number of elements in each pixel's line of sight. The red line shows the actual number of elements necessary in each pixel (*num_los*). The magenta line shows the allocated memory for each pixel using the old memory management, while the green line shows the total allocated memory using the new memory management. As can be seen, almost everywhere the green line is closer to the red line than the magenta line. This is specially true for the large values of *num_los*, where the waste of memory is largest. In other words, the curve of the allocated memory using the new code is much closer to the curve of the actually used memory than was the case in the old code. The memory waste is proportional to the difference in the area below the magenta/green and the red curve, which is clearly smaller for the green than for the magenta curve.

Chapter 5

Optimization of the hierarchical data structure

As discussed in Sec. 2.2.1, the *SPEV* preprocessor structures its output (preprocessed) file into blocks. These blocks contain the information about the non-thermal particles and are accessed by the postprocessor in order to produce the virtual image (see Sec. 4.1). *SPEV* uses the HDF5 library and data format for its preprocessed files. In Sec. 5.1 more details on the structure are given, and some possible problems with the current structure are discussed. Sections 5.2 and 5.3 explain how the structure has been modified and how it has changed the code performance.

5.1 Structure of the preprocessed file

The original structure of the preprocessed file consists of a large number of HDF5 groups, each representing a block of particles. In each group particle data is stored in a number of datasets. The names of the groups are ten-digit numbers, each number corresponding to a block saved by the preprocessor. Some of the problems with this approach are:

- Once the output file is written, sometimes a user might want to access a subset of the data in a particular block using the tool *h5dump*. In runs with high resolution, the size of the file can be in tens of gigabytes. The time it takes to open the file and read the data using *h5dump* may be quite long (minutes or longer).
- The way the data is stored and accessed is quite inflexible, and it is very difficult to access subsets of groups.

With these issues in mind, we have proceeded to modify the file structure to give it more flexibility and to attempt to improve the efficiency.

5.2 Modifications of the file structure

All HDF5 operations which SPEV uses are implemented in a module *hdfman.F*. There, the creation, the opening and the closing of files is implemented, as well as the creation, opening and closing of groups inside files. Further subroutines include the possibility to write and read scalars, vectors and matrices of arbitrary dimensions.

We have added new routines to *hdfman.F* to make the creation and opening of groups more flexible. The idea is to create a hierarchy of groups, and to make this hierarchy easily configurable. In the original code, the identifier of each group is a ten-digit number, and the possible values range from 0000000000 to 9999999999. In other words, all groups are stored on a single level. This idea is illustrated in the top-left box in Fig. 5.1.

After the modification, we can choose to store groups in several levels (at least one, at most ten). This is achieved by splitting the digits of the group identifier among different levels. In the following list we give some examples. The total number of digits is 10, and the items in the list show how they are distributed among different levels for, for example, 612837 blocks of data:

- A single level (original code):
  10 digits, identifiers range from 0000000000 to 0000612837
- Two levels (shown are digits for the upper and lower level):

  _

	Level 2	Level 1
digits	7	3
identifier start	0000000	000
identifier end	0000612	999

  _

	Level 2	Level 1
digits	6	4
identifier start	000000	0000
identifier end	000061	9999

  _

	Level 2	Level 1
digits	5	5
identifier start	00000	00000
identifier end	00006	99999

  _

	Level 2	Level 1
digits	3	7
identifier start	000	0000000
identifier end	000	0612837

- Three levels:

	Level 3	Level 2	Level 1
digits	4	3	3
identifier start	0000	000	000
identifier end	0000	612	999

	Level 3	Level 2	Level 1
digits	5	3	2
identifier start	00000	000	00
identifier end	00006	999	99

	Level 3	Level 2	Level 1
digits	6	2	2
identifier start	000000	00	00
identifier end	000061	99	99

- Four levels:

	Level 4	Level 3	Level 2	Level 1
digits	4	2	2	2
identifier start	0000	00	00	00
identifier end	0000	61	99	99

	Level 4	Level 3	Level 2	Level 1
digits	5	2	2	1
identifier start	00000	00	00	0
identifier end	00006	99	99	9

- Ten levels:

  - 1 - 1 - 1 - 1 - 1 - 1 - 1 - 1 - 1 - 1

    Level 10: identifiers from 0 to 9
    Level 9: identifiers from 0 to 9
    Level 8: identifiers from 0 to 9
    Level 7: identifiers from 0 to 9
    Level 6: identifiers from 0 to 9
    Level 5: identifiers from 0 to 9
    Level 4: identifiers from 0 to 9
    Level 3: identifiers from 0 to 9
    Level 2: identifiers from 0 to 9
    Level 1: identifiers from 0 to 9

The Fig. 5.1 shows a diagram with some of the examples of group hierarchy.

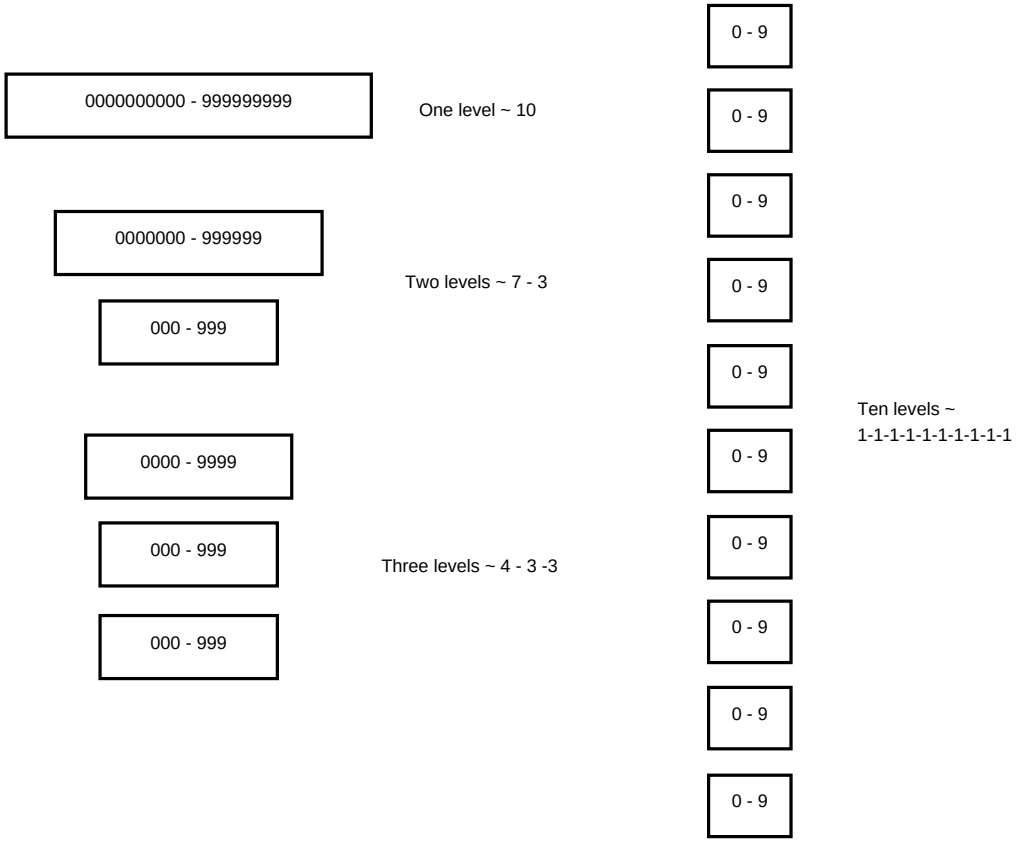

Figure 5.1: Illustration of different group hierarchies. The top-left box shows the original implementation, consisting of just one level which stores all groups. On the right side we see the opposite extreme, where each digit of the group identifier is stored on a different level. The second and third boxes on the left side show a hierarchy with two leves, where first 7 digits are stored on the top, and the last 3 digits on the bottom level. The bottom three boxes show the same for a case with three levels.

5.2.1 Implementation

We have added two new routines which open and close a hierarchy of groups.

The routine *hm_gopen_h* opens or creates a new group hierarchy determined by a predefined format. If there is no format provided, the routine falls back to the default setting (one level), and is, thus, compatible with the files created by the previous version of the code. The format is provided as an array of 10 integers, denoting the number of digits for each level. The convention is that levels with zero digits are not used, and that the lowest level is always at the position 10. This means that zeros in the format array (if any) are always at the lowest positions. Finally, the total sum of all elements in the format array must be 10, and no element can have a value less

than 0 or greater than 10.

The routine calculates the names of all groups in the hierarchy once both the format and the number of the current block has been provided. If some group in the hierarchy does not exist, it is created; otherwise it is opened. In the Fig. 5.2 we see the flow chart of this routine.

The routine *hm_gclose_h* closes a hierarchy of opened groups.

The source code of the *hm_gopen_h* and *hm_gclose_h* routines can be found in Appendix 1.2.

5.3 Performance tests

We have run a set of tests to check if there are improvements in the time it takes to read the data using the *h5dump* tool. The tests have been performed for different levels of hierarchy and the configurations are (showing number of digits at each level):

- 1 level: 0-0-0-0-0-0-0-0-0-10
- 2 levels: 0-0-0-0-0-0-0-0-7-3
- 4 levels: 0-0-0-0-0-0-1-1-4-4
- 10 levels: 1-1-1-1-1-1-1-1-1-1

We performed a test using *h5dump -H*:

	1 Level	2 Levels	4 Levels	10 Levels
AVG TIME	35,05s	37,15s	38,44s	42,50s

The next test is the reading of the last stored block, 23976. The particular arguments to *h5dump* depend on the level hierarchy:

- 1 level:*h5dump -d /0000023976/time pre-0-0-0-0-0-0-0-0-0-10.h5*
- 2 levels: *h5dump -d /0000023/976/time pre-0-0-0-0-0-0-0-0-7-3.h5*
- 4 levels: *h5dump -d /0/0/0002/3976/time pre-0-0-0-0-0-0-1-1-4-4.h5*
- 10 levels: *h5dump -d /0/0/0/0/0/2/3/9/7/6/time pre-1-1-1-1-1-1-1-1-1-1.h5*

	1 Level	2 Levels	4 Levels	10 Levels
AVG TIME	14,27s	15,39s	16,84s	20,63s

The following is the same test, but now reading the first block:

- 1 level: *h5dump -d /0000000001/time pre-0-0-0-0-0-0-0-0-0-10.h5*
- 2 levels: *h5dump -d /0000000/001/time pre-0-0-0-0-0-0-0-0-7-3.h5*

- 4 levels: *h5dump -d /0/0/0000/0001/time pre-0-0-0-0-0-0-1-1-4-4.h5*
- 10 levels: *h5dump -d /0/0/0/0/0/0/0/0/0/1/time pre-1-1-1-1-1-1-1-1-1-1.h5*

	1 Level	2 Levels	4 Levels	10 Levels
AVG TIME	14,47s	15,65s	16,96s	20,53s

Finally, a test reading the block 11988, which is an intermediate value:

- 1 level: *h5dump -d /0000011988/time pre-0-0-0-0-0-0-0-0-0-10.h5*
- 2 levels: *h5dump -d /0000011/988/time pre-0-0-0-0-0-0-0-0-7-3.h5*
- 4 levels: *h5dump -d /0/0/0001/1988/time pre-0-0-0-0-0-0-1-1-4-4.h5*
- 10 levels: *h5dump -d /0/0/0/0/0/1/1/9/8/8/time pre-1-1-1-1-1-1-1-1-1-1.h5*

	1 Level	2 Levels	4 Levels	10 Levels
AVG TIME	14,34s	15,41s	16,81s	20,57s

The results of these tests show that the time needed to access the data in the original one-level structure is smaller than when using more complex hierarchies. Therefore, we did not achieved the goal of improving the time, but we have implemented a more flexible file format which allows the user to configure the levels at will.

We have tested the execution time in the postprocessing process of *SPEV* in order to check if this time is affected by the modified routine of reading the input file with hierarchical levels. The input files of these tests are the output files of preprocessing process of *SPEV* used in tests above

- 1 level: pre-0-0-0-0-0-0-0-0-0-10.h5
- 2 levels: pre-0-0-0-0-0-0-0-0-7-3.h5
- 4 levels: pre-0-0-0-0-0-0-1-1-4-4.h5
- 10 levels: pre-1-1-1-1-1-1-1-1-1-1.h5

	1 Level	2 Levels	4 Levels	10 Levels
AVG TIME	280,6s	278s	279,3s	283,6s

These results confirm that the modifications don't increase the execution time, and the impact on the overall code performance are smaller than a 5%. Therefore, the modifications will remain in the code because we have obtained a more flexible configurations of file format.

In a future work, we will continue studying the improvement of this issue by looking for new routines or parameter values for the routines in HDF5 library. We also think that the file system could influence in this improvement.

In the future we plan to check if the modifications in hierarchical levels have better results in a parallel file system like GPFS.

Furthermore, in the current preliminary study, we have not considered the impact of a more flexible output file structure on the writting time of each file, which is non-negligible for codes dumping datasets with sized between hundreds of Gigabytes to few Terabytes.

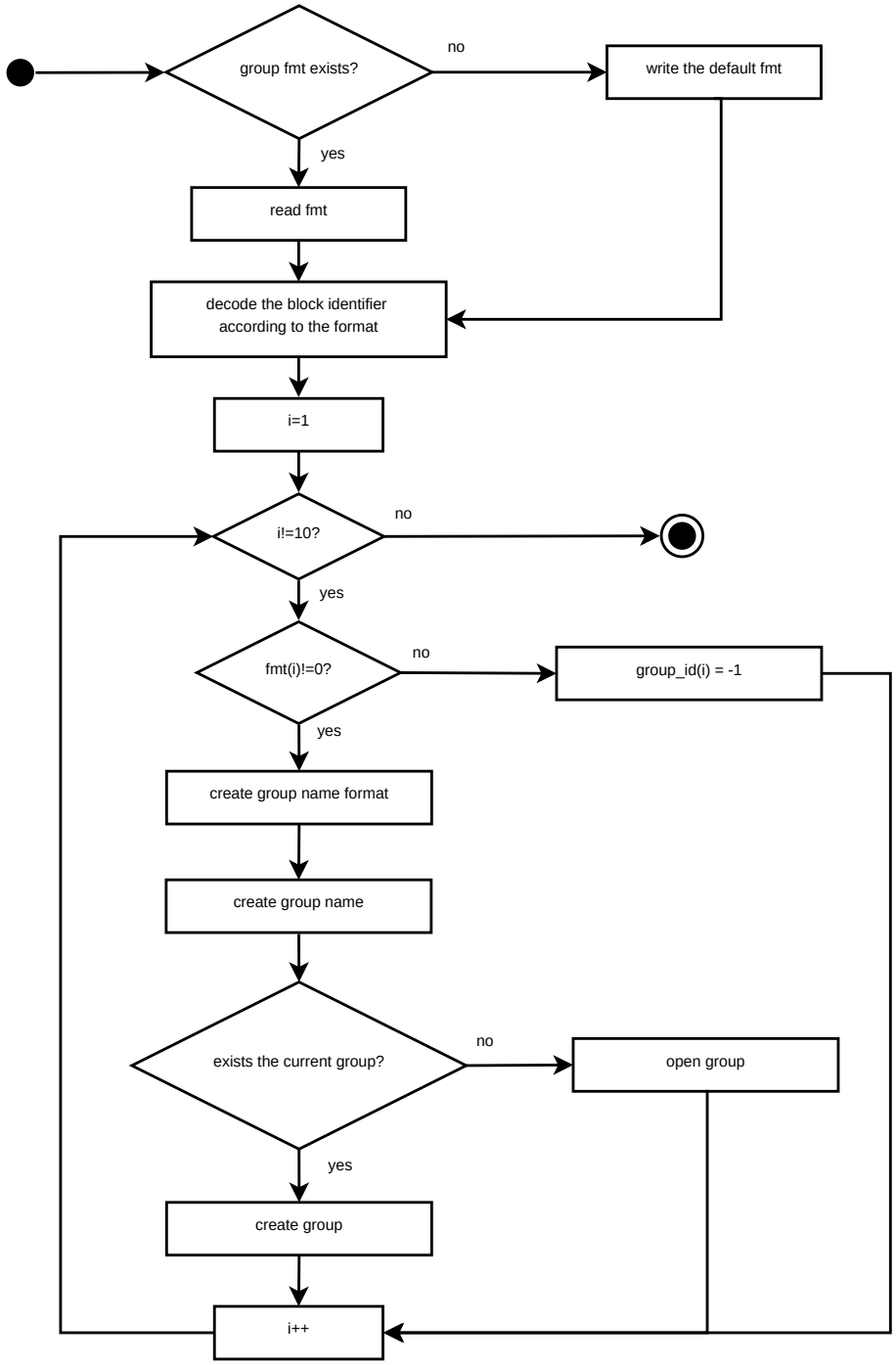

Figure 5.2: Flow chart of the modifications to hdfman.F. The routine *hm_gopen_h* opens or creates a new group depending on the predefined hierarchy. If there is no predefined hierarchy, it uses a single level by default (as in the previous version). Then it calculates the names of all levels. If a group does not exist, it is created; otherwise it is opened.

Chapter 6

Conclusions

The aim of this work is the optimization of the numerical tools which are essential for the research conducted by members of Relativistic Astrophysics Group (RAG) at the University of Valencia. The motivation for code optimization is fully explained in the Introduction. In summary the main reasons for optimization are:

1. Reducing the economic costs of using supercomputers;
2. Reducing the environmental impact, reducing energy consumption by machines and facilities; and
3. to access large European supercomputing facilities, with the purpose of carrying out world leading numerical simulations in Astrophysics, which needed of a large number of processors and use large amounts of memory.

In this work we have focused on *SPEV*, a numerical code which can compute the spectral evolution of non-thermal particles as they undergo synchrotron loses and/or adiabatic transformations (because of the change in the underlying plasma properties), and simultaneously, it can produce synthetic light curves and/or images of the considered problem (Fig. 2). In other words, *SPEV* can produce *synthetic* observations (virtual images) based on RMHD numerical models. These can be compared directly with actual Astrophysical observations. The motivation for *SPEV* optimization is discussed in Sec. 2.1.

We have studied the best configuration of input parameters and implemented several optimizations in order to improve the code efficiency. We optimized according to these parameters:

- speed of execution;
- memory consumption;
- disk usage; and

- network bandwidth.

In this thesis we undertook the following steps to achieve these optimizations in the key parts of the execution of this parallel code:

1. *Preparation of work environment*: This is the beginning of the work. We install the compilers, wrappers and libraries necessary to compile and execute the codes. It has to be kept in mind that the codes run in parallel over shared and distributed memory architectures. We found the best configuration of versions of all these tools so that our codes run efficiently. At the end of this task, we implement an automatic installation script, independent of the platform in wich these tools will be installed. The scientific members of the group will be able to easily install all these tools, needing only to run the script on their own computers, without a need for deep knowledge of system administration. With this advance, they will not need to invest time worrying about technical details.

2. *Profiling and analysis of the computational aspect of* SPEV: In this phase we have discussed the basic algorithm of the *SPEV* code and performed the profiling of its two main components (preprocessor and postprocessor). In Sec. 2.1 we have described the *SPEV* algorithm and given reasons for its optimization. Finally, in Sec. 2.2 we have performed extensive profiling and testing of various aspects of the *SPEV* code. The conclusions after studying the results of the tests of both preprocessor and postprocessor point towards the fact that it makes sense to use the compression when writing preprocessed files. However, the preprocessor in general has to be executed only once, while the postprocessor will be executed multiple times. Furthermore, the execution time of the postprocessor is, in general, longer than that of the preprocessor. Therefore, we find that the best option for *SPEV* is to choose block size of 50000 and a compression level 9.

3. *Optimization of use of hardware resources*: We have modified the codes *MRGENESIS* and *SPEV* to run side-by-side on a multiprocessor machine. In addition, we modified the *PARPLOT* shell script, which contains the code that produces images from the RMHD simulations. This script was adapted to simultaneous execution with *MRGENESIS* too. With these changes we have achieved a speed up of 2.07 in a low resolution execution and a speed up of 1.15 of a high resolution execution, compared to the case when the three codes have to be executed in sequence. In the high resolution case, for a typical 5-hour run we have saved at least 40 minutes. We note that with increasing resolution the savings in time can be significantly smaller.

4. *Optimization of memory management*: As was discussed in Sec. 2.1, and as can be seen in Fig. 2.1, *SPEV* postprocessor uses significant amounts of RAM during its execution. Furthermore, the memory space is allocated dynamically, it grows during the execution and the total required memory capacity *cannot be calculated in advance*. To improve the memory management, we have implemented linked lists. They allow us to append new nodes during the execution, thus allocating memory more efficiently. The results of the tests shows that the results produced by the old code are identical to ones produced by the new, as is explained in Sec. 4.2.3. We have demonstrated the improvement of the efficiency of memory management in Sec. 4.2.3, where it is verified that the efficiency is approximately 99% in most of cases, and can be up to 31% higher than the efficiency of the previous version.

5. *Optimization of hierarchical data structure*: As discussed in Sec. 2.2.1, the *SPEV* preprocessor structures its output (preprocessed) file into blocks. These blocks contain the information about the non-thermal particles and are accessed by the postprocessor in order to produce the virtual image (see Sec. 4.1). *SPEV* uses the HDF5 library and data format for its preprocessed files. In Sec. 5.1 more details on the structure are given. A possible problem with the old structure is when a user wants to access a subset of the data in a particular block using the tool *h5dump*. In runs with high resolution, the size of the file can be in tens of gigabytes. The time it takes to open the file and read the data using *h5dump* may be quite long (minutes or longer). In Sec. 5.2 we have explained how the structure has been modified and how it has changed the code. After implementing several tests to check the improvement ins Sec. 5.3, we concluded that the results of these tests show that the time needed to access the data in the original one-level structure is a bit better than when using more complex hierarchies. Therefore, we did not achieve the goal of improving access times, but we have implemented a more flexible file format which allows the user to configure the levels at will.

After performing all the optimizations we can conclude that the updates implemented have achieved most of the goals proposed in the present work:

- Speed of execution: In Chap. 2 we found the best configuration of input parameters for the *SPEV* preprocessor that produces faster execution in realistic use cases. Furthermore, we have reduced the total time of its execution. This was achieved with the implementation of modifications to execute concurrent tasks explained in Chap. 3. The results of this work have been used to improve the efficiency of the overal code in [6]

- Memory consumption: We have achieved over 99% efficiency in memory management by implemented linked lists, as is demonstrated in Chap. 4.
- Disk usage and Bandwidth requirements: We found the best configuration of block sizes and of the level of compression of the output files in preprocessing part of *SPEV*, so that smaller output files are produced. As a consequence, the bandwidth required to transfer the files from supercomputers where the code is executed to the local computers of the users is significantly reduced and, also, the disk usage is optimized due to the reduction of the file sizes. The results obtained performing this task are presented, in part, in [9] and [16]

In a future work, we will continue studying and improving *SPEV* code with regards to its memory and disk space requirements. We will also study further improvements of writing and reading files in HDF5 format by looking for new routines or parameter values of routines in HDF5 library. We also think that the file system could influence in this improvement, so we plan to check if the modifications in hierarchical levels have better results in a parallel file system like GPFS.

Appendix A

Source Code

1.1 Implementation of Linked Lists

The linked lists are implemented in a separated file `list_losdata.F` into a module, where are written all needed routines to manage these lists. The implementation of the full module is:

```
MODULE list_losdata
     use vars
     IMPLICIT NONE
     PUBLIC

     TYPE node_dist
        DOUBLEPRECISION, DIMENSION(:, :), ALLOCATABLE :: dist
        type(node_dist), pointer :: next
        INTEGER :: fres, lres
     END TYPE node_dist

     TYPE node_emiss
        DOUBLEPRECISION, DIMENSION(:, :), ALLOCATABLE :: emiss
        type(node_emiss), pointer :: next
        INTEGER :: fres, lres
     END type node_emiss

     TYPE node_absor
        DOUBLEPRECISION, DIMENSION(:, :), ALLOCATABLE :: absor
        type(node_absor), pointer :: next
        INTEGER :: fres, lres
     END TYPE node_absor

     TYPE los_elem
        type (node_dist), pointer :: list_dist
        type (node_dist), pointer :: first_dist
        type (node_emiss), pointer :: list_emiss
        type (node_emiss), pointer :: first_emiss
        type (node_absor), pointer :: list_absor
        type (node_absor), pointer :: first_absor
        INTEGER*8 :: totalsize_dist, totalsize_emiss, totalsize_absor, totalsize
```

```
      END TYPE los_elem

      TYPE (los_elem), DIMENSION(:, :, :), ALLOCATABLE :: los_data

      TYPE node_arrays
          DOUBLEPRECISION, DIMENSION(:), ALLOCATABLE :: dist, emiss, absor
      END TYPE node_arrays

      TYPE(node_arrays), DIMENSION(:), ALLOCATABLE :: arrays
      PUBLIC arrays

      CONTAINS

      SUBROUTINE allocate_temporary(num_threads, fres, lres)
      IMPLICIT NONE
      INTEGER :: num_threads, fres, lres, i
      allocate(arrays(num_threads))

      DO i = 1, num_threads
         allocate(arrays(i)%dist(1:2))
         allocate(arrays(i)%emiss(1:fres))
         allocate(arrays(i)%absor(1:fres))
      ENDDO
      END SUBROUTINE allocate_temporary

      SUBROUTINE deallocate_temporary(num_threads)
      IMPLICIT NONE
      INTEGER :: num_threads, i
      DO i = 1, num_threads
         deallocate(arrays(i)%dist)
         deallocate(arrays(i)%emiss)
         deallocate(arrays(i)%absor)
      ENDDO
      deallocate(arrays)
      END SUBROUTINE deallocate_temporary

!     Initialization of dist list
      SUBROUTINE list_init_dist(di, dj, dk)
      IMPLICIT NONE
      INTEGER:: di, dj, dk
      allocate(los_data(di,dj,dk)%list_dist)
      nullify(los_data(di,dj,dk)%list_dist%next)
      los_data(di,dj,dk)%first_dist => los_data(di,dj,dk)%list_dist
      los_data(di,dj,dk)%totalsize_dist = 0
      END SUBROUTINE list_init_dist

!     Initialization of emiss list
      SUBROUTINE list_init_emiss(di, dj, dk)
      IMPLICIT NONE
      INTEGER:: di, dj, dk
      allocate(los_data(di,dj,dk)%list_emiss)
      nullify(los_data(di,dj,dk)%list_emiss%next)
      los_data(di,dj,dk)%first_emiss => los_data(di,dj,dk)%list_emiss
```

```
      los_data(di,dj,dk)%totalsize_emiss = 0
      END SUBROUTINE list_init_emiss

!     Initialization of absor list
      SUBROUTINE list_init_absor(di, dj, dk)
      IMPLICIT NONE
      INTEGER:: di, dj, dk
      allocate(los_data(di,dj,dk)%list_absor)
      nullify(los_data(di,dj,dk)%list_absor%next)
      los_data(di,dj,dk)%first_absor => los_data(di,dj,dk)%list_absor
      los_data(di,dj,dk)%totalsize_absor = 0
      END SUBROUTINE list_init_absor

!     Current node size of dist list
      INTEGER FUNCTION nodeSize_dist(current)
      IMPLICIT NONE
      TYPE(node_dist), pointer  :: current
      nodeSize_dist = current%lres
      END FUNCTION nodeSize_dist

!     Current node size of emiss list
      INTEGER FUNCTION nodeSize_emiss(current)
      IMPLICIT NONE
      TYPE(node_emiss), pointer  :: current
      nodeSize_emiss = current%lres
      END FUNCTION nodeSize_emiss

!     Current node size of absor list
      INTEGER FUNCTION nodeSize_absor(current)
      IMPLICIT NONE
      TYPE(node_absor), pointer  :: current
      nodeSize_absor = current%lres
      END FUNCTION nodeSize_absor

!     Total size of lists
      INTEGER FUNCTION list_totalSize(di, dj, dk)
      IMPLICIT NONE
      INTEGER:: di, dj, dk
      list_totalsize = los_data(di,dj,dk)%totalsize
      END FUNCTION list_totalSize

!     Total size of dist list
      INTEGER FUNCTION list_nodeSize_dist(di, dj, dk)
      IMPLICIT NONE
      INTEGER:: di, dj, dk
      list_nodesize_dist = los_data(di,dj,dk)%totalsize_dist
      END FUNCTION list_nodeSize_dist

!     Total size of emiss list
      INTEGER FUNCTION list_nodeSize_emiss(di, dj, dk)
      IMPLICIT NONE
      INTEGER:: di, dj, dk
      list_nodesize_emiss = los_data(di,dj,dk)%totalsize_emiss
```

```
      END FUNCTION list_nodeSize_emiss

!     Total size of absor list
      INTEGER FUNCTION list_nodeSize_absor(di, dj, dk)
      IMPLICIT NONE
      INTEGER:: di, dj, dk
      list_nodesize_absor = los_data(di,dj,dk)%totalsize_absor
      END FUNCTION list_nodeSize_absor

!     Total size of dist list in bytes
      INTEGER FUNCTION list_totalSize_dist(di, dj, dk)
      IMPLICIT NONE
      INTEGER:: di, dj, dk
!     total elements of array = (totalsize_dist * 2)
!     space occupied is totalsize_dist * 8 bytes
      list_totalsize_dist = los_data(di,dj,dk)%totalsize_dist * 8 * 2
      END FUNCTION list_totalSize_dist

!     Total size of emiss list in bytes
      INTEGER FUNCTION list_totalSize_emiss(di, dj, dk)
      IMPLICIT NONE
      INTEGER:: di, dj, dk
!     total elements of array = (totalsize_dist * fres)
      list_totalsize_emiss = los_data(di, dj, dk)%totalsize_emiss * 8 * fres
      END FUNCTION list_totalSize_emiss

!     Total size of absor list in bytes
      INTEGER FUNCTION list_totalSize_absor(di, dj, dk)
      IMPLICIT NONE
      INTEGER:: di, dj, dk
!     total elements of array = (totalsize_dist * fres)
      list_totalsize_absor = los_data(di,dj,dk)%totalsize_absor * 8 * fres
      END FUNCTION list_totalSize_absor

!     Copy all elements of all nodes of dist list into a temporary array
      SUBROUTINE lists_readAllElem_dist(di, dj, dk, num_elems, out_dist)
      IMPLICIT NONE
      INTEGER, INTENT(IN) :: di, dj, dk, num_elems
      DOUBLEPRECISION, DIMENSION(1:2, 1:num_elems), INTENT(INOUT) :: out_dist
      TYPE(node_dist), pointer :: current
      INTEGER :: i, nodesize
      current => los_data(di, dj, dk)%first_dist
      nodesize = nodeSize_dist(current)
      DO i = 1, num_elems
         IF(i.gt.1.and.mod(i - 1, nodesize).eq.0) current => current%next
         out_dist(1:2, i) = current%dist(1:2, mod(i - 1, nodesize) + 1)
      ENDDO
      RETURN
      END SUBROUTINE lists_readAllElem_dist

!     Copy all elements of all nodes of emiss list into a temporary array
      SUBROUTINE lists_readAllElem_emiss(di, dj, dk, num_elems, out_emiss)
      IMPLICIT NONE
```

```
      INTEGER, INTENT(IN) :: di, dj, dk, num_elems
      DOUBLEPRECISION, DIMENSION(1:fres, 1:num_elems), INTENT(INOUT) :: out_emiss
      type(node_emiss), pointer :: current
      INTEGER :: i, nodesize

      current => los_data(di, dj, dk)%first_emiss
      nodesize = nodeSize_emiss(current)
      DO i = 1, num_elems
         IF(i.gt.1.and.mod(i - 1, nodesize).eq.0) current => current%next
         out_emiss(1:fres, i) = current%emiss(1:fres, mod(i - 1, nodesize) + 1)
      ENDDO
      RETURN
      END SUBROUTINE lists_readAllElem_emiss

!     Copy all elements of all nodes of absor list into a temporary array
      SUBROUTINE lists_readAllElem_absor(di, dj, dk, num_elems, out_absor)
      IMPLICIT NONE
      INTEGER, INTENT(IN) :: di, dj, dk, num_elems
      DOUBLEPRECISION, DIMENSION(1:fres, 1:num_elems), INTENT(INOUT) :: out_absor
      TYPE(node_absor), pointer :: current
      INTEGER :: i, nodesize

      current => los_data(di, dj, dk)%first_absor
      nodesize = nodeSize_absor(current)
      DO i = 1, num_elems
         IF(i.gt.1.and.mod(i - 1, nodesize).eq.0) current => current%next
         out_absor(1:fres, i) = current%absor(1:fres, mod(i - 1, nodesize) + 1)
      ENDDO
      RETURN
      END SUBROUTINE lists_readAllElem_absor

!     Get value of "i" element of all lists for
!     element di,dj,dk of los_data for the current OpenMP thread
      SUBROUTINE lists_getElem(di, dj, dk, i, cur_thread)
      IMPLICIT NONE
      INTEGER:: di, dj, dk, i, cur_thread
      call list_getElem_dist  (di, dj, dk, i, cur_thread)
      call list_getElem_emiss (di, dj, dk, i, cur_thread)
      call list_getElem_absor (di, dj, dk, i, cur_thread)
      END SUBROUTINE lists_getElem

!     Get value of "i" element of dist list for
!     element di,dj,dk of los_data for the current OpenMP thread
      SUBROUTINE list_getElem_dist(di, dj, dk, i, cur_thread)
      IMPLICIT NONE
      INTEGER:: di, dj, dk, i, ind, numnode, nodesize, cnt, cur_thread
      REAL:: ceil
      TYPE(node_dist), pointer  :: current

      cnt=1
      IF( associated(los_data(di,dj,dk)%first_dist) ) THEN
         current => los_data(di,dj,dk)%first_dist
         nodesize = nodeSize_dist(current)
```

```
         numnode = (i - 1 - mod(i - 1, nodesize)) / nodesize + 1
         DO WHILE (numnode.gt.cnt)
            IF(associated(current%next)) THEN
               current => current%next
               cnt = cnt + 1
            ELSE
               print*, 'Error reading list dist (di, dj, dk, i, cnt) = ', di, dj, dk, i, cnt
               STOPINFORM
               STOP
            ENDIF
         ENDIF
         ind = mod (i - 1, nodesize) + 1
         arrays(cur_thread)%dist(:) = current%dist(:,ind)
      ELSE
         STOPINFORM
         STOP 'Error, list does not exist '
      ENDIF
      END SUBROUTINE list_getElem_dist

!     Get value of "i" element of emiss list for
!     element di,dj,dk of los_data for the current OpenMP thread
      SUBROUTINE list_getElem_emiss(di, dj, dk, i, cur_thread)
      IMPLICIT NONE
      INTEGER:: di, dj, dk, i, ind, numnode, nodesize, cnt, cur_thread
      REAL:: ceil
      TYPE(node_emiss), pointer  :: current

      cnt=1
      IF( associated(los_data(di,dj,dk)%first_emiss) ) THEN
         current => los_data(di,dj,dk)%first_emiss
         nodesize = nodeSize_emiss(current)
         numnode = (i - 1 - mod(i - 1, nodesize)) / nodesize + 1
         DO WHILE (numnode.gt.cnt)
            IF(associated(current%next)) THEN
               current => current%next
               cnt = cnt + 1
            ELSE
               print*, 'Error reading list emiss (di, dj, dk, i, cnt) = ', di, dj, dk, i, cnt
               STOPINFORM
               STOP
            ENDIF
         ENDIF
         ind = mod (i - 1, nodesize) + 1
         arrays(cur_thread)%emiss(:) = current%emiss(:,ind)
      ELSE
         STOPINFORM
         STOP 'Error, list does not exist '
      ENDIF
      END SUBROUTINE list_getElem_emiss

!     Get value of "i" element of absor list for
!     element di,dj,dk of los_data for the current OpenMP thread
      SUBROUTINE list_getElem_absor(di, dj, dk, i, cur_thread)
```

```
      IMPLICIT NONE
      INTEGER:: di, dj, dk, i, ind, numnode, nodesize, cnt, cur_thread
      REAL:: ceil
      TYPE(node_absor), pointer  :: current

      cnt=1
      IF( associated(los_data(di,dj,dk)%first_absor) ) THEN
         current => los_data(di,dj,dk)%first_absor
         nodesize = nodeSize_absor(current)
         numnode = (i - 1 - mod(i - 1, nodesize)) / nodesize + 1
         DO WHILE (numnode.gt.cnt)
            IF(associated(current%next)) THEN
               current => current%next
               cnt = cnt + 1
            ELSE
               print*, 'Error reading list absor (di, dj, dk, i, cnt) = ', di, dj, dk, i, cnt
               STOPINFORM
               STOP
            ENDIF
         ENDDO
         ind = mod (i - 1, nodesize) + 1
         arrays(cur_thread)%absor(:) = current%absor(:,ind)
      ELSE
         STOPINFORM
         STOP 'Error, list does not exist '
      ENDIF
      END SUBROUTINE list_getElem_absor

!     Set value of "i" element of all lists for
!     element di,dj,dk of los_data for the current OpenMP thread
      SUBROUTINE lists_setElem(di, dj, dk, i, cur_thread)
      IMPLICIT NONE
      INTEGER:: di, dj, dk, i, cur_thread
      call list_setElem_dist  (di, dj, dk, i, cur_thread)
      call list_setElem_emiss (di, dj, dk, i, cur_thread)
      call list_setElem_absor (di, dj, dk, i, cur_thread)
      END SUBROUTINE lists_setElem

!     Set value of "i" element of dist list for
!     element di,dj,dk of los_data for the current OpenMP thread
      SUBROUTINE list_setElem_dist(di, dj, dk, i, cur_thread)
      IMPLICIT NONE
      INTEGER:: di, dj, dk, i, ind, numnode, nodesize, cnt, cur_thread
      REAL:: ceil
      TYPE(node_dist), pointer  :: current

      cnt=1
      IF(.not. associated(los_data(di,dj,dk)%first_dist) ) THEN
         call list_append_dist(di, dj, dk, fres, lres)
      ENDIF

      current => los_data(di,dj,dk)%list_dist
      nodesize = nodeSize_dist(current)
```

```
      IF(i.gt.1.and.mod(i - 1, nodesize).eq.0) THEN
         call list_append_dist(di, dj, dk, current%fres, current%lres)
         current => current%next
         cnt = cnt  + 1
      ENDIF
      ind = mod (i - 1, nodesize) + 1
      current%dist(:,ind) = arrays(cur_thread)%dist(:)
      END SUBROUTINE list_setElem_dist

!     Set value of "i" element of emiss list for
!     element di,dj,dk of los_data for the current OpenMP thread
      SUBROUTINE list_setElem_emiss(di, dj, dk, i, cur_thread)
      IMPLICIT NONE
      INTEGER:: di, dj, dk, i, ind, numnode, nodesize, cnt, cur_thread
      REAL:: ceil
      TYPE(node_emiss), pointer  :: current

      cnt=1
      IF(.not. associated(los_data(di,dj,dk)%first_emiss) ) THEN
         call list_append_emiss(di, dj, dk, fres, lres)
      ENDIF

      current => los_data(di,dj,dk)%list_emiss
      nodesize = nodeSize_emiss(current)
      IF(i.gt.1.and.mod(i - 1, nodesize).eq.0) THEN
         call list_append_emiss(di, dj, dk, current%fres, current%lres)
         current => current%next
         cnt = cnt  + 1
      ENDIF
      ind = mod (i - 1, nodesize) + 1
      current%emiss(:,ind) = arrays(cur_thread)%emiss(:)
      END SUBROUTINE list_setElem_emiss

!     Set value of "i" element of absor list for
!     element di,dj,dk of los_data for the current OpenMP thread
      SUBROUTINE list_setElem_absor(di, dj, dk, i, cur_thread)
      IMPLICIT NONE
      INTEGER:: di, dj, dk, i, ind, numnode, nodesize, cnt, cur_thread
      REAL:: ceil
      TYPE(node_absor), pointer  :: current

      cnt=1
      IF(.not. associated(los_data(di,dj,dk)%first_absor) ) THEN
         call list_append_absor(di, dj, dk, fres, lres)
      ENDIF

      current => los_data(di,dj,dk)%list_absor
      nodesize = nodeSize_absor(current)
      IF(i.gt.1.and.mod(i - 1, nodesize).eq.0) THEN
         call list_append_absor(di, dj, dk, current%fres, current%lres)
         current => current%next
         cnt = cnt  + 1
      ENDIF
```

```
      ind = mod (i - 1, nodesize) + 1
      current%absor(:,ind) = arrays(cur_thread)%absor(:)
      END SUBROUTINE list_setElem_absor

!     Append a new node in all lists for
!     element di,dj,dk of los_data for the current OpenMP thread
      SUBROUTINE lists_append(di, dj, dk, fres, lres)
      IMPLICIT NONE
      INTEGER :: di, dj, dk, fres, lres
      call list_append_dist  (di, dj, dk, fres, lres)
      call list_append_emiss (di, dj, dk, fres, lres)
      call list_append_absor (di, dj, dk, fres, lres)
      END SUBROUTINE lists_append

!     Append a new node in dist list for
!     element di,dj,dk of los_data for the current OpenMP thread
!     The size of new node is 2*lres
      SUBROUTINE list_append_dist(di, dj, dk, fres, lres)
      IMPLICIT NONE
      INTEGER, intent(in) ::  di, dj, dk, fres, lres

      IF( associated(los_data(di,dj,dk)%first_dist) ) THEN
         allocate(los_data(di,dj,dk)%list_dist%next)
         los_data(di,dj,dk)%list_dist => los_data(di,dj,dk)%list_dist%next
         nullify(los_data(di,dj,dk)%list_dist%next)
      ELSE
!     is first
         call list_init_dist(di,dj,dk)
      ENDIF

      los_data(di,dj,dk)%list_dist%lres = lres
      los_data(di,dj,dk)%list_dist%fres = fres
      los_data(di,dj,dk)%totalsize_dist = los_data(di,dj,dk)%totalsize_dist + lres
      los_data(di,dj,dk)%totalsize = los_data(di,dj,dk)%totalsize_dist
      allocate(los_data(di,dj,dk)%list_dist%dist(1:2, 1:lres))
      END SUBROUTINE list_append_dist

!     Append a new node in emiss list for
!     element di,dj,dk of los_data for the current OpenMP thread
!     The size of new node is fres*lres
      SUBROUTINE list_append_emiss(di, dj, dk, fres, lres)
      IMPLICIT NONE
      INTEGER, intent(in) ::  di, dj, dk, fres, lres

      IF( associated(los_data(di,dj,dk)%first_emiss) ) THEN
         allocate(los_data(di,dj,dk)%list_emiss%next)
         los_data(di,dj,dk)%list_emiss => los_data(di,dj,dk)%list_emiss%next
         nullify(los_data(di,dj,dk)%list_emiss%next)
      ELSE
!     is first
         call list_init_emiss(di,dj,dk)
      ENDIF
```

```
      los_data(di,dj,dk)%list_emiss%lres=lres
      los_data(di,dj,dk)%list_emiss%fres=fres
      los_data(di,dj,dk)%totalsize_emiss = los_data(di,dj,dk)%totalsize_emiss + lres
      los_data(di,dj,dk)%totalsize = los_data(di,dj,dk)%totalsize_emiss
      allocate(los_data(di,dj,dk)%list_emiss%emiss(1:fres, 1:lres))
      END SUBROUTINE list_append_emiss

!     Append a new node in absor list for
!     element di,dj,dk of los_data for the current OpenMP thread
!     The size of new node is fres*lres
      SUBROUTINE list_append_absor(di, dj, dk, fres, lres)
      IMPLICIT NONE
      INTEGER, intent(in) ::  di, dj, dk, fres, lres

      IF( associated(los_data(di,dj,dk)%first_absor) ) THEN
         allocate(los_data(di,dj,dk)%list_absor%next)
         los_data(di,dj,dk)%list_absor => los_data(di,dj,dk)%list_absor%next
         nullify(los_data(di,dj,dk)%list_absor%next)
      ELSE
!     is first
         call list_init_absor(di,dj,dk)
      ENDIF

      los_data(di,dj,dk)%list_absor%lres=lres
      los_data(di,dj,dk)%list_absor%fres=fres
      los_data(di,dj,dk)%totalsize_absor = los_data(di,dj,dk)%totalsize_absor + lres
      los_data(di,dj,dk)%totalsize = los_data(di,dj,dk)%totalsize_absor
      allocate(los_data(di,dj,dk)%list_absor%absor(1:fres, 1:lres))
      END SUBROUTINE list_append_absor

!     Destroy all lists for element di,dj,dk of los_data
      SUBROUTINE lists_destroy(di, dj, dk)
      IMPLICIT NONE
      INTEGER :: di, dj, dk
      call list_destroy_dist(di, dj, dk)
      call list_destroy_emiss(di, dj, dk)
      call list_destroy_absor(di, dj, dk)
      END SUBROUTINE lists_destroy

!     Destroy dist list for element di,dj,dk of los_data
      SUBROUTINE list_destroy_dist(di, dj, dk)
      IMPLICIT NONE
      INTEGER :: di, dj, dk
      DO WHILE (associated(los_data(di,dj,dk)%first_dist))
         deallocate(los_data(di,dj,dk)%first_dist%dist)
         los_data(di,dj,dk)%first_dist => los_data(di,dj,dk)%first_dist%next
      ENDDO
      END SUBROUTINE list_destroy_dist

!     Destroy emiss list for element di,dj,dk of los_data
      SUBROUTINE list_destroy_emiss(di, dj, dk)
      IMPLICIT NONE
      INTEGER :: di, dj, dk
```

```
      DO WHILE (associated(los_data(di,dj,dk)%first_emiss))
         deallocate(los_data(di,dj,dk)%first_emiss%emiss)
         los_data(di,dj,dk)%first_emiss => los_data(di,dj,dk)%first_emiss%next
      ENDDO
      END SUBROUTINE list_destroy_emiss

!     Destroy absor list for element di,dj,dk of los_data
      SUBROUTINE list_destroy_absor(di, dj, dk)
      IMPLICIT NONE
      INTEGER :: di, dj, dk
      DO WHILE (associated(los_data(di,dj,dk)%first_absor))
         deallocate(los_data(di,dj,dk)%first_absor%absor)
         los_data(di,dj,dk)%first_absor => los_data(di,dj,dk)%first_absor%next
      ENDDO
      END SUBROUTINE list_destroy_absor

END MODULE list_losdata
```

1.2 Implementation of the hierarchical data structure

The source code of the `hm_gopen_h` and `hm_gclose_h` routines is the following:

```
!     create HDF5 groups with hierarchical levels
      SUBROUTINE hm_gopen_h(file_id, num_id, group_ids)
      use hdf5
      IMPLICIT NONE
      INTEGER(HID_T), INTENT(INOUT) :: file_id
      INTEGER(HID_T), INTENT(INOUT), DIMENSION(10) :: group_ids
      INTEGER, INTENT(IN) :: num_id
      INTEGER, DIMENSION(10) :: fmt, id_fmt
      INTEGER :: i, j, current_num, current_base
      INTEGER(HID_T) :: last_id
      CHARACTER(len=256) :: group_name, group_digits

!     read file format
      IF (hm_exists(file_id, "fmt")) THEN
         CALL hm_read1_int(file_id, 10, fmt, "fmt")
      ELSE
         fmt = (/0, 0, 0, 0, 0, 0, 0, 0, 0, 10/)
      ENDIF

!     decode num_id according to the format
      current_num = num_id
      DO i = 10, 1, -1
         IF (fmt(i).ne.0) THEN

!     compute current base
            current_base = 1
```

```
            DO j = 1, fmt(i)
               current_base = current_base * 10
            ENDDO

!     current number is modulo current base
            id_fmt(i) = mod(current_num, current_base)

!     new number is the integer part of the division
            current_num = (current_num  - mod(current_num, current_base)) / current_base
         ELSE
            id_fmt(i) = -1
         ENDIF
      ENDDO

!     create group hierarchy
      last_id = file_id
      DO i = 1, 10
         IF (fmt(i).ne.0) THEN

!     create group name format
            IF (fmt(i).lt.10) THEN
               WRITE(group_digits, "(i1.1)") fmt(i)
            ELSE
               WRITE(group_digits, "(i2.2)") fmt(i)
            ENDIF

!     create group name
            WRITE(group_name, "(i"//TRIM(group_digits)//"."//TRIM(group_digits)//")") id_fmt(i)

!     open or create the current group
            IF (hm_exists(last_id, TRIM(group_name))) THEN
               call hm_gopen(last_id, TRIM(group_name), group_ids(i))
            ELSE
               call hm_gcreate(last_id, TRIM(group_name), group_ids(i))
            ENDIF
            last_id = group_ids(i)
         ELSE
            group_ids(i) = -1
         ENDIF
      ENDDO
      RETURN
      END SUBROUTINE hm_gopen_h

!     close HDF5 groups with hierarchical levels
      SUBROUTINE hm_gclose_h(group_ids)
      use hdf5
      IMPLICIT NONE
      INTEGER(HID_T), DIMENSION(10), INTENT(INOUT) :: group_ids
      INTEGER :: i

      DO i = 10, 1, -1
         IF (group_ids(i).ne.-1) call hm_gclose(group_ids(i))
      ENDDO
```

```
RETURN
END SUBROUTINE hm_gclose_h

END MODULE hdfman
```

Appendix B

Sampling figures

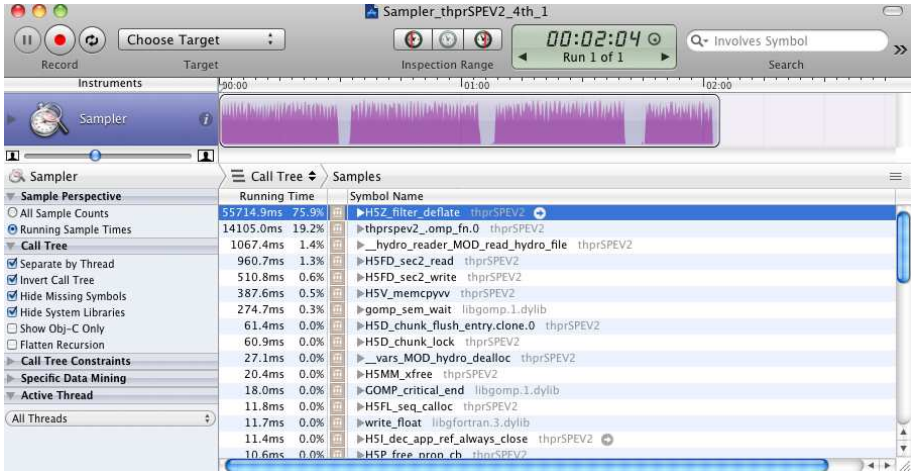

Figure B.1: Test: Compression level = 1. 4 OpenMP threads.

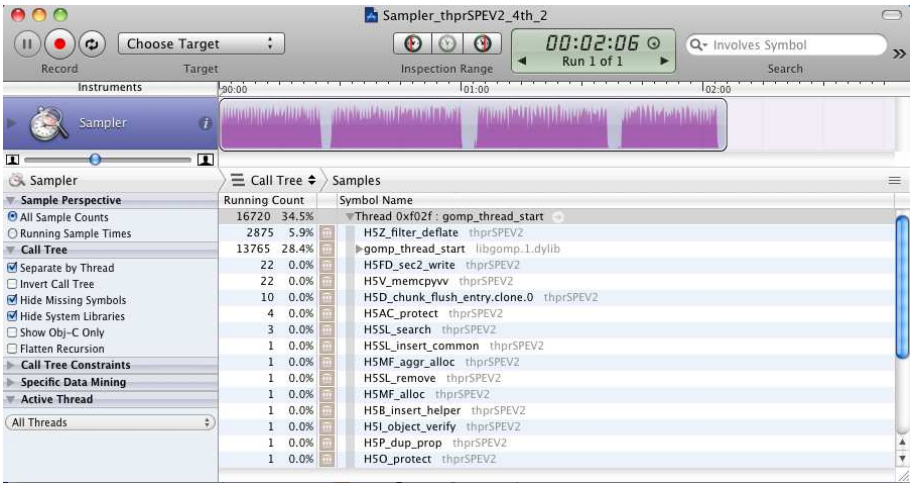

Figure B.2: Test: Compression level = 2. 4 OpenMP threads.

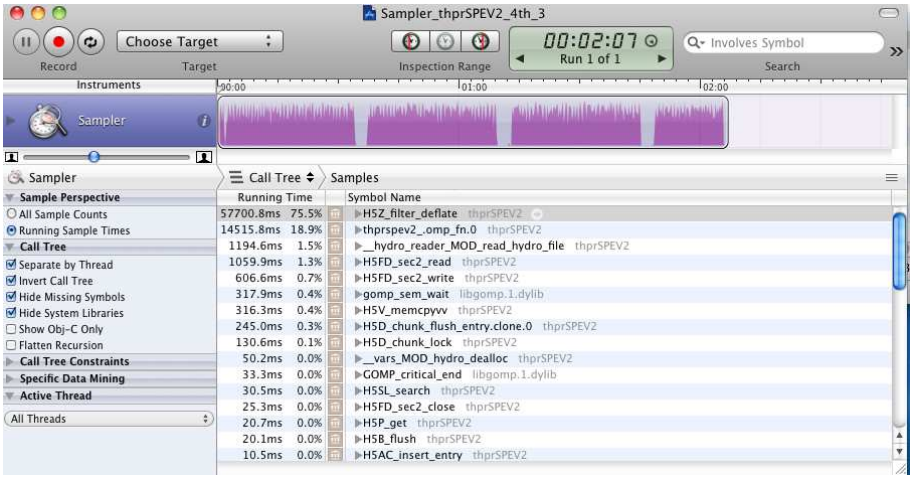

Figure B.3: Test: Compression level = 3. 4 OpenMP threads.

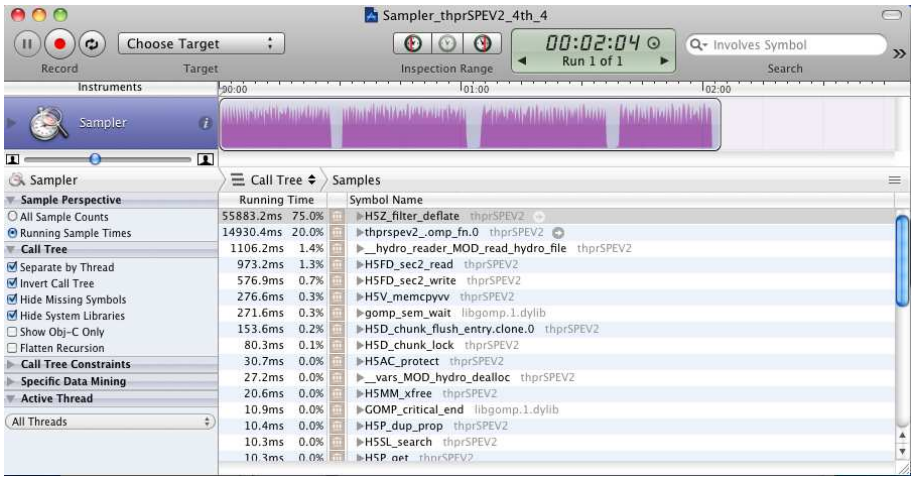

Figure B.4: Test: Compression level = 4. 4 OpenMP threads.

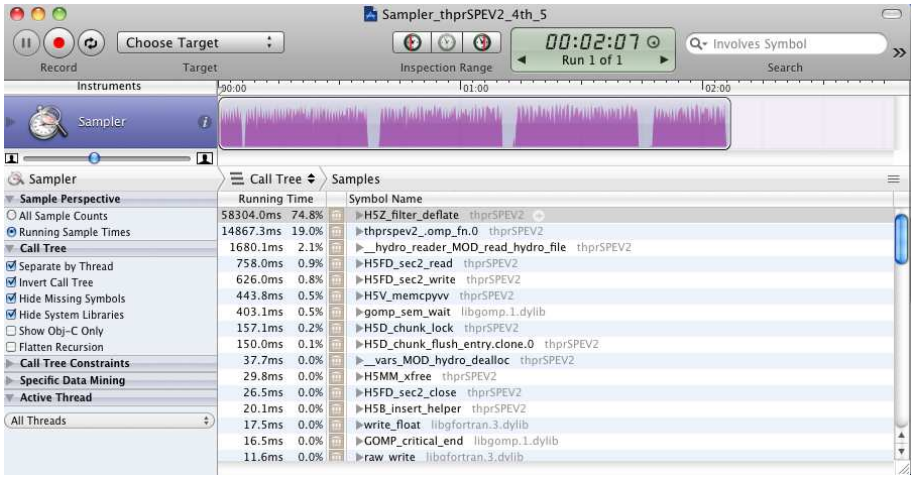

Figure B.5: Test: Compression level = 5. 4 OpenMP threads.

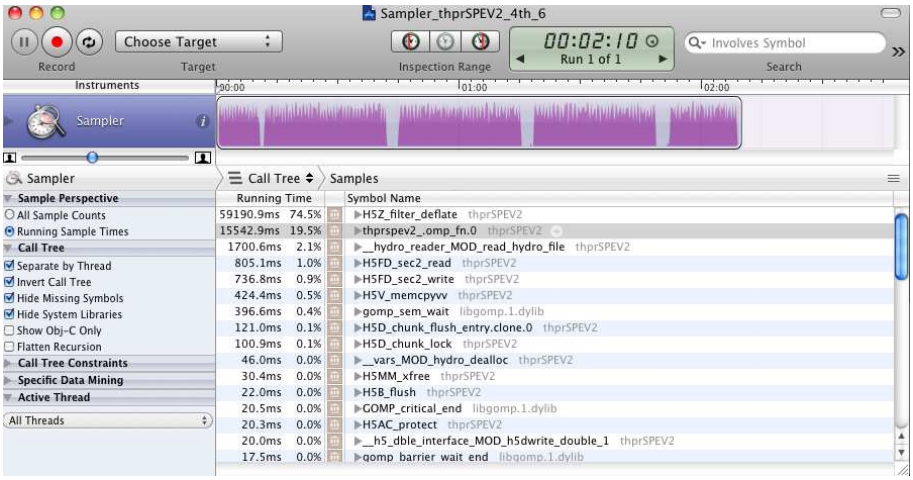

Figure B.6: Test: Compression level = 6. 4 OpenMP threads.

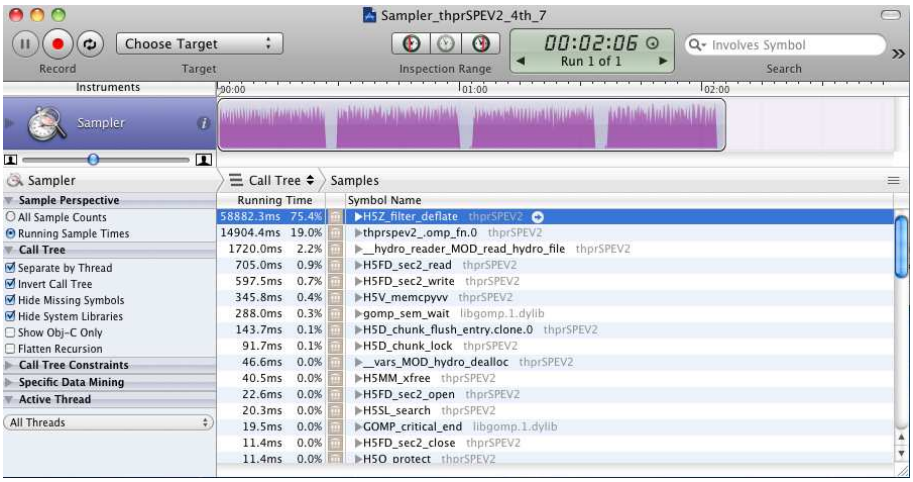

Figure B.7: Test: Compression level = 7. 4 OpenMP threads.

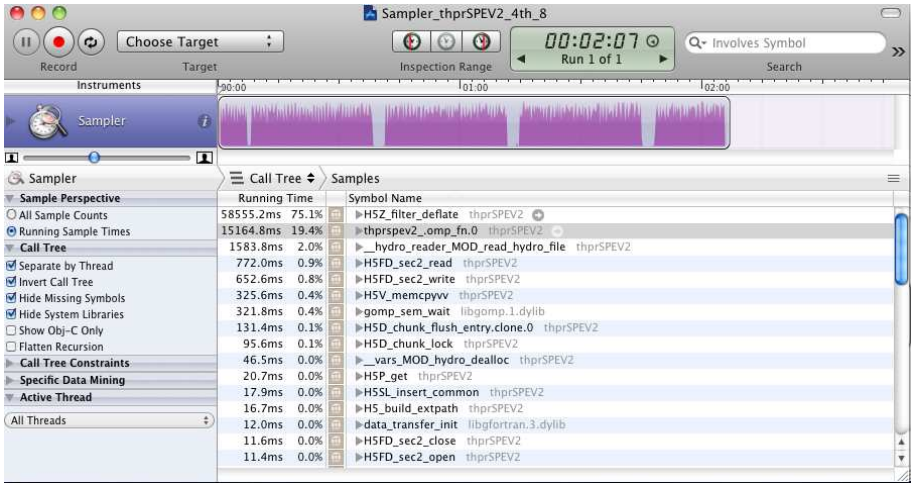

Figure B.8: Test: Compression level = 8. 4 OpenMP threads.

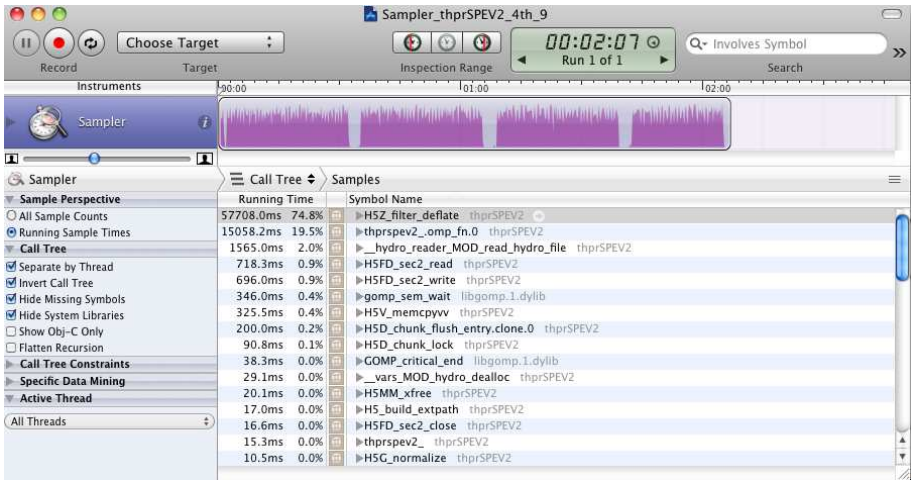

Figure B.9: Test: Compression level = 9. 4 OpenMP threads.

Appendix C

Publications related to the present work

We have published several articles, posters, proceedings and presentations related to the present work. These publications are:

- Article [11]:

  Petar Mimica, Miguel A Aloy, Jesus Rueda-Becerril, Siham Tabik, Carmen Aloy. Numerical simulations of dynamics and emission from relativistic astrophysical jets. *Journal of Physics Conference Series.* 454 012001 doi:10.1088/1742-6596/454/1/012001

  `http://iopscience.iop.org/1742-6596/454/1/012001`

- Presentation [2]:

  Carmen Aloy. Parallel performance improvement in MRGENESIS and RATPENAT. Presentation at Mini-WORKSHOP on Supercomputing and GRID. October 5th, 2012. University of Valencia - IVICFA.

  `http://ivicfa.uv.es/?page_id=341#SGrid`

  Slides available at: `http://bit.ly/14XMLDI`

- Poster [6]:

  Miguel A Aloy, Sergio Miranda, Carmen Aloy. Building a non-ideal numerical relativistic magnetohydrodynamics code for astrophysical applications. Poster at ASTRONUM Conference. July 1st - July 5th, 2013, Biarritz (France).

  Available at `http://www.uv.es/macarato/posterASTRONUM.pdf`.

  Conference: `http://irfu.cea.fr/Projets/ASTRONUM2013/index.htm`

- Proceedings [9]:

  Petar Mimica, Miguel A Aloy, Carlos Cuesta and Carmen Aloy. Challenges in computing thermal and non-thermal emission from relativistic

outflows Presentation at ASTRONUM Conference. July 1st - July 5th, 2013, Biarritz (France).

`http://irfu.cea.fr/Projets/ASTRONUM2013/Presentations/tableau.htm`

- Poster and Proceedings [16]:

  Jesús Rueda, Petar Mimica, Miguel A Aloy, Carmen Aloy. Numerical study of broadband spectra caused by internal shocks in magnetized relativistic jets of blazars. Poster at Conference: The Innermost Regions of Relativistic Jets and Their Magnetic Fields. June 10th -14th, 2013. Granada (Spain).

  Available at `http://www.uv.es/macarato/posterGRANADA.pdf`.

  Link to the Conference: `http://jets2013.iaa.es`

  The proceedings of this poster has been sent in September 11th, 2013, to be published. Link to the abstract `http://bit.ly/1aN9tTp`

Bibliography

[1] C. Aloy, P. Mimica, and M. A. Aloy. Poster presented at PRACE Summer School, CSC, Helsinki (Finland), 2011. Available online at `http://www.uv.es/macarato/poster.pdf`.

[2] C. Aloy, P. Mimica, and M. A. Aloy. Parallel performance improvement in MRGENESIS and RATPENAT. Presentation at Mini-WORKSHOP on Supercomputing and GRID. University of Valencia, 2012. Available online at `http://bit.ly/14XMLDI`.

[3] M. A. Aloy, J. M. Ibáñez, J. M. Martí, and E. Müller. GENESIS: A High-Resolution Code for Three-dimensional Relativistic Hydrodynamics. *The Astrophysical Journal Supplement Series*, 122:151–166, May 1999.

[4] M. A. Aloy and P. Mimica. *Simulations of Jets from Active Galactic Nuclei and Gamma-Ray Bursts;* in *Relativistic Jets from Active Galactic Nuclei (eds M. Bttcher, D. E. Harris and H. Krawczynski)*, chapter 10, pages 297–339. Wiley-VCH Verlag GmbH & Co. KGaA, 2012.

[5] M. A. Aloy, P. Mimica, I. Agudo, J. M. Marti, J. L. Gómez, and J. A. Miralles. Spectral Evolution of Superluminal Components in Parsec-Scale Jets. *The Astrophysical Journal*, 696(2):1142–1163, May 2009.

[6] M. A. Aloy, S. Miranda, and C. Aloy. Building a non-ideal numerical relativistic magnetohydrodynamics code for astrophysical applications. Poster at ASTRONUM Conference. July 1st - July 5th, 2013, Biarritz-France. Available online at `http://www.uv.es/macarato/posterASTRONUM.pdf`.

[7] S. N. Borovikov, N. V. Pogorelov, G. P. Zank, and I. A. Kryukov. Consequences of the Heliopause Instability Caused by Charge Exchange. *The Astrophysical Journal*, 682(2):1404–1415, 2008.

[8] Heerikhuisen J. and Pogorelov, N. V. An Estimate of the Nearby Interestellar Magnetic Field Using Neutral Atoms. *The Astrophysical Journal*, 738(1):29–38, 2011.

[9] P. Mimica, M. A. Aloy, C. Cuesta, and C. Aloy. Challenges in computing thermal and non-thermal emission from relativistic outflows. Proceedings at ASTRONUM Conference. July 1st - July 5th, 2013, Biarritz-France. Available online at `http://irfu.cea.fr/Projets/ASTRONUM2013/Presentations/tableau.htm`.

[10] P. Mimica, M. A. Aloy, and E. Müller. Internal shocks in relativistic outflows: collisions of magnetized shells. *Astronomy & Astrophysics*, 466:93–106, April 2007.

[11] P. Mimica, M. A. Aloy, J. Rueda, S. Tabik, and C. Aloy. Numerical simulations of dynamics and emission from relativistic astrophysical jets. *Journal of Physics: Conference Series*, 454(1):012001, 2013.

[12] Pogorelov, N. V. and Stone, E. C. and Florinski, V. and Zank, G. P. Termination Shock Asymmetries as Seen by the Voyager Spacecraft: The Role of the Interstellar Magnetic Field and Neutral Hydrogen. *The Astrophysical Journal*, 688(2):661–624, 2006.

[13] Pogorelov, N. V. and Zank, G. P. and Ogino, T. Three-dimensional Features of the Outer Heliosphere Due to Coupling between the Interstellar and Interplanetary Magnetic Fields. I. Magnetohydrodynamic Model: Interstellar Perspective. *The Astrophysical Journal*, 614(2):1007–1021, 2004.

[14] Pogorelov, N. V. and Zank, G. P. and Ogino, T. Three-dimensional Features of the Outer Heliosphere due to Coupling between the Interstellar and Interplanetary Magnetic Fields. II. The Presence of Neutral Hydrogen Atoms. *The Astrophysical Journal*, 644(2):1299–1316, 2006.

[15] PRACE. HPC access. How to apply, 2012. Available online at `http://www.prace-ri.eu/How-to-apply?lang=en`.

[16] J. Rueda, P. Mimica, M. A. Aloy, and C. Aloy. Numerical study of broadband spectra caused by internal shocks in magnetized relativistic jets of blazars. Poster at Conference: The Innermost Regions of Relativistic Jets and Their Magnetic Fields. June 10th -14th, 2013. Granada (Spain). Available online at `http://www.uv.es/macarato/posterGRANADA.pdf`.

[17] D. Vicente. BSC-RES-PRACE, HPC resources, 2012. Available online at `http://indico.ific.uv.es`.