Technische Universität München

Lehrstuhl für Informatik mit Schwerpunkt Wissenschaftliches Rechnen

Enabling efficient simulations of radiative transfer with CRASH

Nitya Hariharan

Vollständiger Abdruck der von der Fakultät für Informatik Technischen Universität München zur Erlangung des akademischen Grades eines

Doktors der Naturwissenschaften (Dr. rer. nat.)

genehmigten Dissertation.

Vorsitzender:		Univ.-Prof. Dr. Hans Michael Gerndt
Prüfer der Dissertation:	1.	Univ.-Prof. Dr. Hans-Joachim Bungartz
	2.	Hon.-Prof. Simon D.M. White, Ph.D. Ludwig-Maximilians-Universität München

Die Dissertation wurde am 09.04.2015 bei der Technischen Universität München eingereicht und durch die Fakultät für Informatik am 15.06.2015 angenommen.

Zusammenfassung

Der rasante Fortschritt in der Hochleistungsrechnung ermöglicht heute das Lösen von einer Fülle an wissenschaftlichen Problemen. Von groëssem Nutzen ist dies auch im Bereich der Astrophysik, wo, sowohl die Theorie als auch die Beobachtungen, die Wissenschaft voran treiben. Die Entwicklung effizienter numerischer Methoden hat es ermöglicht, umfangreiche kosmologische Simulationen durchzuführen, welche uns Einblick in die Frühphase des Universums gewähren, und seine Entwicklung bis hin zur Gegenwart. Diese Simulationen helfen auch dabei, theoretische Modelle mit Messdaten abzugleichen und anhand von Beobachtungen zu überprüfen. Um ein vollständiges Bild des Universums darstellen zu können, muss eine große Anzahl physikalischer Abläufe berücksichtigt werden, welche auf vielschichtige Weise miteinander wechselwirken und sich über einige räumliche sowie zeitliche Größenordnungen erstrecken.

Diese Prozesse, allgemein eingestuft als chemische, mechanische oder strahlungs- Rückkopplung, bilden eine wichtige Komponente kosmologischer Simulationen. Ihre akkurate Behandlung in der numerischen Simulation ist rechentechnisch sehr aufwändig und setzt einige Annahmen voraus, welche die Komplexität des Problems verringern. Unentwegt wird versucht, die grossskaligen Simulationen durch die Entwicklung besserer numerischer Methoden zu verbessern, sodass die zur Verfügung stehenden Rechenresourcen möglichst effizient genutzt werden.

Speziell der Mechanismus der Strahlungsrückkopplung ist dabei von Bedeutung, auf Grund seines sowohl kurzreichweitigen als auch fernwirksamen Einflusses. Er wird mit Hilfe von Strahlungs-Transfer (RT) Verfahren numerisch dargestellt. Diese Doktorarbeit befasst sich mit einem solchen Verfahren, genannt `CRASH`. Es handelt sich um einen 3D Monte Carlo RT Code, welcher selbstkonsistent die Entwicklung von Wasserstoff und Helium simuliert, sowohl in neutraler als auch in ionisierter Form, gemeinsam mit der Temperatur des Gas Mediums. Der Code wird als Postprocessing Tool für kosmologische hydrodynamische Simulationen verwendet, um die Auswirkung von Strahlungsrückkopplung auf Strukturbildung und Reionisierung des intergalaktischen Mediums zu untersuchen. Die Fähigkeit, hochauflösende Simulationen mit `CRASH` durchzuführen, ist daher eine entscheidende Voraussetzung. Diese Art von Simulationen sind derzeit mit `CRASH` allerdings nicht möglich auf Grund seiner algorithmischen Einschränkungen.

Um die Leistungsfähigkeit von `CRASH` zu verbessern und seine Verwendung auszuweiten, wird die Benutzung ineinander verschachtelte, verfeinerte Gitter ermöglicht. Der erste wichtige Schritt ist die Entwicklung einer neuen Version, `CRASH-AMR`. Diese kann RT Simulationen auf verfeinerten Gittern ausführen, wobei die Adaptivität durch einen hydrodynamischen Code

bestimmt wird. CRASH erhält eine Schnittstelle zur Open-Source AMR Bibliothek CHOMBO, und alle Software Entwicklungen in CRASH-AMR werden eingehend erläutert. CRASH-AMR wird vollständig mit Hilfe einer vorgefertigten standard Testsuite für RT Codes geprüft. Zudem werden hochauflösende RT Simulationen unter Verwendung von realistischen Dichtefeldern durchgeführt, die von einem durch AMR unterstützten Hydrocode erhalten wurden. Dadurch wird die Fähigkeit von CRASH-AMR getestet, Veränderungen im Gradienten des Gasdichtefeldes korrekt zu verfolgen, welche die Strahlungs-Materie Wechselwirkung in der RT Simulation beeinflusst.

Als Erweiterung der Arbeit wird CRASH-AMR parallelisiert mittels MPI und der relevanten Grundstruktur im Rahmen von CHOMBO. Diese Arbeit enthält eine detaillierte Erläuterung der verwendeten Parallelisierungstechniken. Der Code wird mit Hilfe von standard RT Testfällen geprüft. Ausserdem wird die Leistungsfähigkeit des Codes unter Einsatz eines realistischen Dichtefeldes betrachtet. Dabei werden verschiedene Verfeinerungsstufen des Gitters und zwei unterschiedliche Maschinen des Rechenzentrums Garching verwendet.

Nun kann CRASH-AMR als Postprocessing Tool für einen durch AMR unterstützten Hydrocode verwendet werden. Die Einführung von Adaptivität in den RT Code selbst ist jedoch schon ein Schritt nach vorn in der adaptiven Gitterverfeinerung mittels Kriterien, welche durch RT Simulationen gegeben werden. Auch wird die Durchführbarkeit der Implementierung einer dynamischer Gitterverfeinerung in CRASH-AMR im Rahmen von AMR in CHOMBO betrachtet und ein relevantes Beispiel untersucht.

Mit diesen wesentlichen Erweiterungen ist es jetzt möglich, hochauflösende RT Simulationen mit CRASH durchzuführen; die neuen Entwicklungen bilden die Grundlage einer Kopplung des Codes mit einem kosmologischen hydrodynamischen Code, um umfangreiche Simulationen zur Strukturbildung durchzuführen.

Abstract

The rapid advances in computing have made it possible for a wide range of scientific problems to be addressed. This has been especially useful in the area of astrophysics, that relies on theories and observations for its progress. The development of efficient numerical methods has made it possible to carry out large-scale cosmological simulations, giving us an insight into the early stages of the universe and its evolution to the present day. These simulations have also helped to verify and validate theoretical models against observational data. To provide a complete picture of the universe, the inclusion of a large number of physical processes that interact in a complex manner is required. These are relevant over spatial and temporal scales that span several orders of magnitude.

These processes, broadly classified as chemical, mechanical and radiative feedback, form an important aspect of cosmological simulations. Their precise treatment in numerical simulations is computationally very expensive and necessitates that certain assumptions be made in order to reduce their complexity. Efforts are constantly underway to improve these large-scale simulations by developing better numerical methods that can make efficient use of the computational resources available.

In particular, the radiative feedback mechanism is of importance due to its short and long range effects and is modelled using radiative transfer (RT) schemes. This thesis focusses on one such scheme, `CRASH`, a 3D Monte Carlo RT code that can self-consistently follow the evolution of H and He, both neutral and ionized species, along with the temperature of the gas medium. The code is used as a post-processing tool for cosmological hydrodynamic simulations to study the effects of radiative feedback on structure formation and reionisation of the intergalactic medium. The ability to perform very high resolution simulations using `CRASH` is hence a crucial requirement. Such kind of simulations, though, are currently not feasible with `CRASH` due to its computational, i.e. algorithmic limitations.

To improve the performance and extend the use of `CRASH`, we enable the use of nested, refined grids. The first major step is the development of a new version, `CRASH-AMR`, that can perform RT simulations on refined grids, whose adaptivity is decided by a hydro code. We interface `CRASH` with the open-source Adaptive Mesh Refinement (AMR) library `CHOMBO`, and discuss in detail the software developments carried out in `CRASH-AMR`. The code has been fully tested against a suite of standard test cases prescribed for RT codes. Also, we carry out high resolution RT simulations using realistic density fields obtained from an AMR enabled hydro code. This tests the capability of `CRASH-AMR` to correctly track the variations in the gas density field gradients that affects the radiation-matter interaction in the RT simulation.

As an extension of our work, we parallelise CRASH-AMR using MPI and the relevant framework available in CHOMBO. A detailed discussion about the techniques used to parallelise the code is given. The code has been tested with the standard RT test cases. We also look at the performance of the code, when RT simulations are done on a realistic density field set on grids with multiple refinement levels, on two different machines at the Rechenzentrum Garching.

CRASH-AMR can now be used as a post-processing tool for an AMR enabled hydro code. The inclusion of adaptivity within the RT code itself is, however, an important step forward in being able to adaptively refine the grid according to criteria as dictated by the RT simulation. We take a look at the feasibility of implementing dynamic refinement in CRASH-AMR using the AMR framework in CHOMBO and study a relevant example.

With these substantial extensions, it is now possible to carry out high-resolution RT simulations with CRASH; the new development sets the stage for coupling the code with a cosmological hydrodynamic code to carry out large scale simulations of structure formation.

Acknowledgements

There are a number of people I wish to thank, hopefully I have mentioned everyone here. My sincere thanks to my supervisor, Benedetta Ciardi, for giving me an opportunity to work on this project and for guiding me so patiently over the years. I am grateful to Professor H. J. Bungartz for accepting me as an external student at his chair, and also for all the support he has given me over the duration of my thesis. Many thanks to Luca Graziani for introducing me to the `CRASH` code, and the problem at hand. He has helped me over the years and I would not have been able to finish my work without his guidance. My thanks to Simon White for accepting me as a PhD candidate at MPA and for his advice and support.

I am for ever grateful to my parents, my brother and his wife for their support over the years; their love and affection has kept me going. My gratitude to all my friends in Munich for making my stay here enjoyable. To all my friends who are not in Munich, thank you for the moral support.

I am thankful to Francesco Miniati and Sebastiano Cantalupo for the valuable discussions I had with them. My special thanks to Cornelia Rickl, Gabi Kratschmann, Maria Depner and Sonja Gruendl for all their help and support at MPA; just talking to them was enough to cheer me up. My gratitude to the system administrators at MPA for providing excellent facilities to work with. I should for sure convey my thanks to Manuela Fischer at TUM, she has patiently cleared all my doubts regarding the thesis submission formalities. And last but not the least, thanks to my amma and appa at Namakkal for their love and blessings.

Contents

Chapter 1

Introduction

The world around us is made of multiple independent or inter-dependent systems. Some of them evolve over a long period of time, for example the very universe we live in that has a history of 13.6 billion years. Or the formation and growth of different layers of rock on the earth's crust that happens over millenia and cannot be witnessed in its entirety by humans per se. Some others can evolve very rapidly, for example the folding of a complex protein molecule, or the flow of fluid in a vortex; these events happen in a short period of time and are hard to understand using mere visual observations.

Theoretical models and experimental set-ups help us to study these phenomena. However, given their complexity, it is not always possible to use analytical models alone. In some situations, experimental techniques might also not give accurate results due to the differences in the experimental set-up and real life conditions, for example studying the air flow around an airplane wing. Some scenarios might have safety concerns, for example the transport of highly inflammable liquids. Others might occur over very long time scales to prevent any reproducibility, such as rock formation or metal corrosion. In such cases, we need to rely on computational techniques to model these systems and to verify our theories and experimental data with simulation data. This is easier said than done since simulating a real world system requires that its analytical model be first described using an algorithm that can then be written as a program on a computer.

The use of computers as tools to model physical systems has undergone a drastic change over the decades. We take a look at the history of computing in this Chapter, which is structured as follows. In Section 1.1 we look at the developments that have taken place in the field of computing in general. The progress of research in many fields has been underpinned by the development of better numerical and simulation techniques and this has also been the case within the astrophysics community. The study of large scale structure formation forms a major aspect of this field and we introduce it briefly in Section 1.2. Then we look at the role of simulations in astrophysics in Section 1.3 and conclude the Chapter with an outline of the thesis in Section 1.4.

1.1 Advances in computing

The beginning of the concept of modern computers, for automated computation, can be traced back to the work of Charles Babbage who designed the Difference Engine in 1849 and the Analytical engine in 1837, which although never built, included the concept of an arithmetic logic unit and control flow for conditional branching; storage for variable values was also provided. This was succeeded by punch card machines designed by Herman Hollerith in 1889. The early and mid 20th century saw rapid changes in computer designs. The development of vacuum tube machines, the ENIAC and EDVAC, established a prototype for the modern computers. Boolean algebra, introduced by George Boole, had a huge impact on information theory and the design of electronic computers. The Turing machine laid the foundations for representing a problem as a set of instructions. This, together with the von Neumann architecture, influenced the design of the first modern computer [101].

In 1960 the first supercomputer, CDC 6600, was manufactured at Control Data Corporation (CDC) by Seymour Cray. The same decade saw the first vector processing machine, the STAR-100, developed at CDC in collaboration with Texas Instruments. Vector machines were the dominant designs for supercomputers through the 70s and 80s, mostly on Cray platforms. In 1990s supercomputers with hundreds or thousands of processors gained prominence that enabled massively parallel computations [101]. These machines could operate with a peak performance of 1 teraFLOPS (floating point operations per second). The last decade has seen a tremendous increase in computing power with the fastest supercomputer of today, Tianhe-2, having a peak performance of 33.8 petaFLOPS [1].

Graphics processing units (GPUs), initially used in computer graphics, now have a considerable presence in the parallel computing scene. The general purpose GPUs of today provide computing power equivalent to the multi-core processors available. Field Programmable Gate Arrays (FPGAs) provide yet another alternative to CPUs and GPUs, they were initially used in digital signal processing but are increasingly in use for scientific and industrial applications, such as oil and gas exploration and cryptography [191].

The advances in computer design and the increase in computing power took place alongside the development of improved numerical and computational methods for simulating complex physical systems. We mention a few notable developments here. Scientific computing as a new research area began in the 1940s, where it was first used in fluid dynamics and by 1950 was also a part of numerical weather predictions. ENIAC which had been designed to calculate ballistic trajectories made use of Ordinary Differential Equations (ODEs) in its calculations. Later in 1950s, the increased memory capacity in computers saw the development of adaptive ODE codes with variable step size [65]. Around the same period, the Conjugate Gradient method was developed [82], a well known iterative method for solving sparse systems of linear equations.

Particle-in-cell methods, used in plasma physics simulations to follow the trajectories of charged particles in an electromagnetic field, were developed in the 1950s. Finite-element (FE) and finite-volume (FV) methods to approximate the solutions of Partial Differential

[1] `http://www.top500.org/system/177999`

Equations (PDEs) became well established in the 60s [132] by the aerospace and civil engineering community [137]; FE methods are now an integral part of numerical analysis in structural mechanics. FV methods, on the other hand, are used in computational fluid dynamics simulations. This decade also saw the Fast Fourier Transform (FFT) gain wide acceptance among numerical analysts [46]. FFT has now been used in a wide range of applications that include spectral analysis, data compression, pattern recognition and for numerical solution of differential equations, to name a few.

The introduction of Smoothed Particle Hydrodynamic (SPH) methods in 1977 [67, 115], to simulate physical processes spanning several orders of magnitude, was another important milestone. The inital application of this method was in astrophysics to non-spherical stars. Eulerian schemes were introduced in [44, 106] but they suffered from the problem of not being able to provide temporal and spatial adaptivity in a simple way. The use of nested grids and Adaptive Mesh Refinement (AMR) techniques introduced in [18, 19] resolved this problem and they became important for simulations that resolved physical effects at much higher resolution.

Nowadays, numerical simulations are accepted as the third pillar of science, along with theory and observations, and are used in almost every field of scientific research. Among these, the field of astrophysics is unique in the sense that one cannot carry out any experiments in the lab to model cosmic objects. Research in this field is driven by observations, theory and simulations which have contributed immensely in understanding the history of the universe. We next discuss large scale structure formation, which forms an important aspect of modern cosmology.

1.2 Large scale structure formation

The Big Bang cosmology is a standard model that explains the primordial stages of the universe. At early epochs, the universe underwent a phase of rapid, exponential expansion called the "cosmic inflation". After the inflationary phase the universe consisted of a hot plasma containing protons, electrons and other light nuclei. Around 400,000 years after the Big Bang the temperature dropped enough to allow protons and electrons to combine and form Hydrogen. This "cosmic recombination" led to the decoupling of radiation and matter, allowing photons to travel further and get redshifted into the microwave regime [13]. This radiation, called the "Cosmic Microwave Background" (CMB), is an important probe of the early universe in observational cosmology. The CMB and its properties have been well understood through the COsmic Background Explorer (COBE) [2], in the 1990s, followed by the Wilkinson Microwave Anisotropy Probe (WMAP) campaign [17] and the recent Planck mission [147]. The anisotropies found in the CMB have allowed us to tightly constrain the cosmological parameters [185].

The universe that we observe today shows a large variation in structures as we move from the large scales of hundreds of Mpc size, to the small, kilo-parsec scales. Independent observations of the distribution of faint radio sources, optically selected galaxies, the X-ray background and the CMB show that our universe is homogeneous on a scale larger than 200 Mpc (see [201]

[2] http://lambda.gsfc.nasa.gov/product/cobe/

for a review), while at scales of tens of Mpc it appears inhomogeneous due to the presence of a large number of structures: from galaxy clusters and galaxies, to stellar clusters and molecular clouds. In the ΛCDM concordance model of our universe, the presence of primordial density perturbations drove the gravitational collapse and cooling of the primordial gas in pre-existing dark-matter (DM) haloes, leading to the formation of radiating sources like stars and quasars [174]. The chemical, mechanical and radiative feedback by these sources on their surroundings induces a complex interplay between the galaxy formation process and the evolution of the intergalactic medium (IGM; [13, 39, 51, 126]).

Chemical feedback is associated to the metal enrichment of the gas by the first generations of star that induces a transition from metal-free Pop III stars to metal-rich Pop II/I stars [118, 165]. Mechanical feedback deals with energy deposition through supernova explosions and galactic winds [161, 173]. Radiative feedback deals with the ionising/dissociating radiation produced by massive stars. Radiative feedback has important consequences, due to its short and long range effects, influencing the evolution of nearby objects and combining with radiation from other sources to form a background [39]. One of the most relevant aspects of radiative feedback is the reionisation of the IGM, which also plays an important role in structure formation. This denotes the transition from a neutral intergalactic gas, to an IGM which is (almost) fully ionised in its hydrogen component by $z \sim 6$ [38, 86], while the helium reionisation is believed to be complete at $z \sim 2.7$ [45, 200].

A number of developments have taken place over the last few decades in computational astrophysics to enable better and faster simulations of the different processes that are involved in the formation and evolution of Large Scale Structures (LSS) in the universe. We discuss them below.

1.3 Role of simulations in astrophysics

The importance of simulations in astrophysics can be made obvious by the fact that the term "Astro-informatics" is now often used, similar to "Geo-informatics" and "Bio-informatics". This refers to the branch of astrophysics that relies on new algorithmic and computational techniques specific to astrophysics [26]. Without simulations it would be impossible to take into account the different physical processes, relevant at different spatial and temporal scales, required to fully understand the universe from the Big Bang to its present state.

The first application of simulations to the study of astrophysical phenomena was done by Martin Schwarzschild, to understand stellar evolution [80]. In 1958 programs to model stellar interiors and stellar atmospheres were discussed by [138] using IBM magnetic drum calculators. Further increase in the computing power available allowed a realistic picture of stellar evolution to emerge [84, 85].

In the 1970s, the very first cosmological N-body simulation, to understand the influence of DM on structure formation, in a comoving periodic cube was done by [129], using 400 particles. This aimed at exploring nonlinear gravitational clustering of objects, and the conclusions reached by the authors were independently verified by [145] and [76]. Following this, in the 90s, a large number of N-body simulations with ever increasing number of particles were carried out

by e.g. [36, 59, 196], [180] (a tree code) and [66] (an adaptive particle-particle-particle mesh code). These simulations helped to firmly establish certain aspects of structure formation, such as abundance and density profiles of DM haloes [181]. The increasing availability of computing resources has enabled N-body simulations of today to reach even larger particle numbers. Examples include the `2HOT` code which is based on the octree method and was run with 128 billion particles [194], `GreeM` which scales up to a trillion particles [89] and the "Millenium XXL" simulation which was done using the `Gadget-3` code with 303 billion particles [6].

In order to fully understand LSS, it is necessary for DM simulations to also include the effects of gas (or hydro) dynamics, feedback effects and star formation. Hydrodynamic simulations can be broadly classified into Eulerian grid-based and Lagrangian methods, such as SPH. Eulerian schemes were used in hydro codes in 1970s to model a rotating protostellar cloud collapse [21, 135]. The SPH method, first used for non-spherical stars as mentioned in Section 1.1, has since been used for a wide range of astrophysical problems such as planetary formation, binary stars and gas dynamics in large-scale cosmological simulations [83].

The use of hierarchical grids that allowed high resolutions only where necessary, was first done by [57] who modeled isolated cloud collapse, including magnetic fields. This was followed by [206] and [117] where they also used hierarchically nested grids to model cloud collapse. Soon AMR hydro codes were developed, which include [159, 184], [30, 198], followed by AMR Magneto Hydrodynamics (MHD) codes, for example [62, 127, 128]. Even though SPH and AMR schemes have helped increase the spatial and temporal scales at which hydrodynamic simulations are done, one still has to make certain assumptions to be able to account for the different physical processes. For this reason, often sub-grid models are used for those effects whose range is smaller than the grid resolution, for example, star formation, feedback from active galactic nuclei (AGN), radiative cooling and photoionisation heating [48, 107, 163].

Cosmological simulations that take into account both DM and hydrodynamics along with sub-grid models have now been carried out. Examples include the EAGLE simulation [164] that was done with the `Gadget-3` code and used 7 billion particles; the Illustris simulation [192] made use of the moving mesh code `Arepo` and reached a total particle count of 18 billion.

The use of GPUs to accelerate codes has also been growing. Examples include simulations of star clusters in galactic nuclei [175], gravitational N-body tree-codes [15, 16] and planetary system simulations [74, 130]. Hybrid CPU/GPU codes are also being developed that work on diverse computing architectures, the Hardware/Hybrid Accelerated Cosmology Code (HACC) was able to perform one of the largest cosmological simulations with 3.6 trillion particles [77].

From our discussion above, it is clear that the field of computational astrophysics involves the use of a large number of processes that act at different spatial scales. The simulation codes currently in use still need to make certain assumptions in order to include all these processes. However, the use of supercomputing facilities has led to major developments that enable the proper inclusion of different phenomena necessary to form a complete picture of LSS. Radiative feedback is one such example that is computationally expensive due to its short and long range effects and also due to its relevance in different environments, such as star forming regions, H II regions and the IGM. A number of different numerical techniques are used to study the effects of the radiation field from sources such as stars and quasars, both independently and along with hydrodynamic simulations. We next discuss the concept of

Radiative Transfer (RT), that is used to describe the radiation field emitted from sources and its propagation along a medium. The numerical formulation of RT along with the techniques used to study it are discussed.

1.3.1 Radiative Transfer

The evolution in space and time of an electromagnetic field as it travels through a medium can be described by the RT equation, which in comoving coordinates is given by [70, 198]

$$\frac{1}{c}\frac{\partial I_\nu}{\partial t} + \frac{\hat{n}\cdot\Delta I_\nu}{\overline{a}} - \frac{H}{c}(\nu\frac{\partial I_\nu}{\partial \nu} - 3I_\nu) = -\kappa_\nu I_\nu + j_\nu \tag{1.1}$$

where the term $I_\nu \equiv I(\nu, x, \Omega, t)$ is the radiation specific intensity in units of energy per time t per solid angle Ω per unit area per frequency ν. $H = \dot{a}/a$ is the Hubble constant where a is the scale factor, $\overline{a} = a/a_{em}$ is the ratio of the scale factors at the current time and the time of emission and $\hat{n}$ is the unit vector in the direction of photon propagation. The second term on the left side describes the propagation of radiation, taking into account the cosmic expansion implied by the factor $1/\overline{a}$. The third term accounts for the dilution of radiation and Doppler redshift of the photons.

On the right hand side, the first term includes κ_ν which is the absorption coefficient $\kappa_\nu \equiv \kappa_\nu(x, \nu, t)$ that describes every frequency dependent event resulting in the absorption of a photon. The second term is the emission coefficient $j_\nu \equiv j_\nu(x, \nu, t)$ that characterises every photon emission process (from point sources or diffuse radiation).

The RT equation, in its original form, is computationally expensive to solve due to its high dimensionality. Some approximations can however be done to reduce its complexity in numerical simulations. The (ν/c) term in Equation 1.1 can be neglected as it applies to relativistic flows or very optically thick systems [198]. The term accounting for cosmic expansion, $\overline{a}$, can be assumed to be 1 since the time steps in dynamic calculations are small enough such that $\Delta a/a \ll 1$. This second term thus reduces to $\hat{n}\partial I_\nu/\partial x$. Finally, the redshifting of photons and dilution of radiation becomes important only for large box sizes, where the light crossing time is comparable to the Hubble time. This can be neglected in the case of a local approximation, thus reducing the RT equation to a much simpler form given by

$$\frac{1}{c}\frac{\partial I_\nu}{\partial t} + \hat{n}\frac{\partial I_\nu}{\partial x} = -\kappa_\nu I_\nu + j_\nu \tag{1.2}$$

Various methods to solve the RT equation, given in Equation 1.2, have been proposed. Each method has its own merits and drawbacks, and is suitable for certain problems relating to RT. Some of them rely on grid based methods while others use the SPH formulation, unstructured grids have also been used to represent the domain over which the RT equations are solved. A number of such RT codes have been compared against a suite of standard tests described in the RT Code Comparison Project (RTCCP) [87]. All the codes tested as part of the comparison project were found to be in good agreement with each other, with some differences arising due to the inherent nature of the method being used to solve the RT equations.

The different techniques available for carrying out RT simulations can be broadly classified into [144]

(a) **Ray-tracing methods** - These methods solve the RT equation along a one dimensional ray that propagates through the medium. These methods can again be classified into three types, depending on how the ray is casted over the cells in the computational domain.

   (i) **Long characteristics method** - In this case, the ray is cast from each source to all the cells in the domain. For each ray that is cast, the optical depth in all the cells lying between the source and the destination cell has to be calculated. This method has a large computational overhead due to the redundant calculations involved [154].

   (ii) **Short characteristics method** - This method removes the disadvantage of the long characteristics method, where the optical depth is calculated multiple times per cell. Here, the individual contribution of the cell to the ray's optical depth is calculated. The total optical depth for each ray originating from a source to a particular cell is determined by summing up and interpolating the contributions of the intermediate cells [154].

   (iii) **Monte Carlo (MC) ray-tracing** - This is a simplification of the long characteristics method where the solution to the RT equation is approximated by emitting rays in random directions away from the source. The photon is propagated along the ray and for each cell crossed by the ray, a set of probability distribution functions that simulate radiation-matter interaction are solved and the properties of the medium are updated. The required convergence of the solution determines the number of photons that need to be emitted from the source [40].

(b) **Variable Eddington tensor** - In this method, the moment equations of the radiation field are solved. This approach ensures that photon numbers and flux is conserved and is also suitable for coupling with grid based hydrodynamic codes. However, the method suffers from the drawback of unphysical propagation of ionisation fronts thus failing to cast a sharp shadow behind an optically thick obstacle [69].

Cosmological RT forms an important part of the LSS research area and we will be focusing on it further in our thesis. We conclude this Chapter by giving an outline of the thesis.

1.4 Thesis outline

The work done as part of this thesis is to develop an enhanced version of the RT code `CRASH`. Our enhancements to the code will allow cosmological RT simulations to be done at much higher resolutions, using AMR techniques, than was previously possible with the code due to its algorithmic limitations.

The thesis is structured as follows, in Chapter 2 we discuss grid generation techniques used to discretise and solve systems of PDEs with focus on AMR techniques. In Chapter 3 we present the RT code `CRASH` explaining in detail the MC scheme used in CRASH to implement RT. The need to do high resolution simulations is an issue for RT simulations as well, this allows them

to evaluate the radiation-matter interaction with much more precision. To this effect, we have developed a new version of the code, CRASH-AMR, that interfaces CRASH with an open source AMR library CHOMBO. This development enables CRASH-AMR to carry out high-resolution RT simulations, in the post-processing mode, on adaptive grids generated by a hydro code. We discuss, in detail, the development of this code; a series of tests validating the new code are provided.

In Chapter 4 we look at parallelising CRASH-AMR using distributed memory parallelism along with tests done to highlight the efficiency of the parallel code. Chapter 5 looks at the feasibility of introducing adaptivity in CRASH-AMR where the grids are refined within the RT simulation itself. Finally we present our conclusions in Chapter 6.

Chapter 2

Adaptive Mesh Refinement

Computers, by their very nature, can only operate on discrete values. Real world systems on the other hand, for example the flow of a fluid, are said to be continuous. Although a fluid is made up of atoms and molecules which interact with each other, on macroscopic scales the fluid motion is said to be continuous. In order to simulate a physical system, we need to define its domain in discrete terms and solve the set of equations that describe the various processes involved. The term grid or mesh is used to refer to this discretised domain. Generating a grid of good quality is of high importance since its ability to correctly track system features can have a huge impact on the solution of the equations, hence grid generation is an important field of research with wide range of applications.

We take a more detailed look at the process of grid generation in this Chapter, which is organised as follows. Section 2.1 looks at how a physical system can be discretised for generating a grid. Depending on the structure of the grid two main categories exist, structured and unstructured. We discuss the different methods employed to generate both types of grids. Simulations of physical systems are, in general, constrained by the grid resolution they use, both in terms of solution accuracy and computational resources. AMR methods have been found to ameliorate this problem to a great extent, by selectively refining the domain in areas that need high resolution. They are used widely across the scientific community including the field of astrophysics, we discuss these methods in detail in Section 2.2.

2.1 Grid generation

Let's consider a system consisting of a viscous fluid. The state of the fluid at any particular point in space and time can be described by a set of properties such as, to name a few, velocity, pressure, viscosity and temperature. We can write the state of the fluid as a continuous function u defined over a set of points in space x and time t where

$$-\infty < x < \infty, t \geq 0 \tag{2.1}$$

The values that x and t can have represent the physical domain over which u is defined. Since u depends on more than one independent variable, i.e. x and t, its rate of change is a Partial Differential Equation (PDE) given by

$$f(x, t, u, u_x, u_t) = 0 \tag{2.2}$$

where the terms u_x and u_t are said to be the partial derivatives of u at points x, t denoted by

$$u_x = \frac{\partial u(x,t)}{\partial x}, u_t = \frac{\partial u(x,t)}{\partial t} \tag{2.3}$$

We can now discretise the domain as

$$U(x, t) = U_0, U_1, U_2, ..., U_N \tag{2.4}$$

by defining the function u at N discrete points $x = -x_N, .., x_0, .., x_N$ and $t = t_0, t_1, ..t_N$ where $N \in \mathbb{R}^+$. The value of the function $U(x, t)$ at some initial and final values of x and t, i.e.

$$U_0 = U(x_0, t_0), \tag{2.5}$$
$$U_N = U(x_N, t_N) \tag{2.6}$$

are the boundary conditions. The values of $U(x, t)$ can now be set on a uniform grid or mesh as shown in the left panel in Figure 2.1. The grid contains rectangular cells that are all of a particular width h. The term width here refers to the change in the value of x or t from one cell to another. The partial derivative with respect to x and t can now be defined as

$$\begin{aligned} \frac{\partial u(x,t)}{\partial x} &= \frac{U(x+1,t) - U(x,t)}{h} \\ \frac{\partial u(x,t)}{\partial t} &= \frac{U(x,t+1) - U(x,t)}{h} \end{aligned} \tag{2.7}$$

For the grid in Figure 2.1, left panel, the values of $U(x, t)$ are set at the center of the cell. Such a grid is said to be cell-centered. We can also set the values at the vertices of the cell, the grid is then said to be vertex-centered. Note that the grid has been discretized both in space and time, since the function $u(x, t)$ that we want to discretize depends on both x and t. This continuous function, now discretised on a grid, can be studied numerically on a computer and forms the computational domain. The physical system that was discretised forms the physical domain. Since we know the extent of the computational domain in x and t, the grid can be stored as an array with the values for $U(x, t)$. The function U can be solved

[1] `http://ta.twi.tudelft.nl/users/wesselin/projects/unstructured.html`

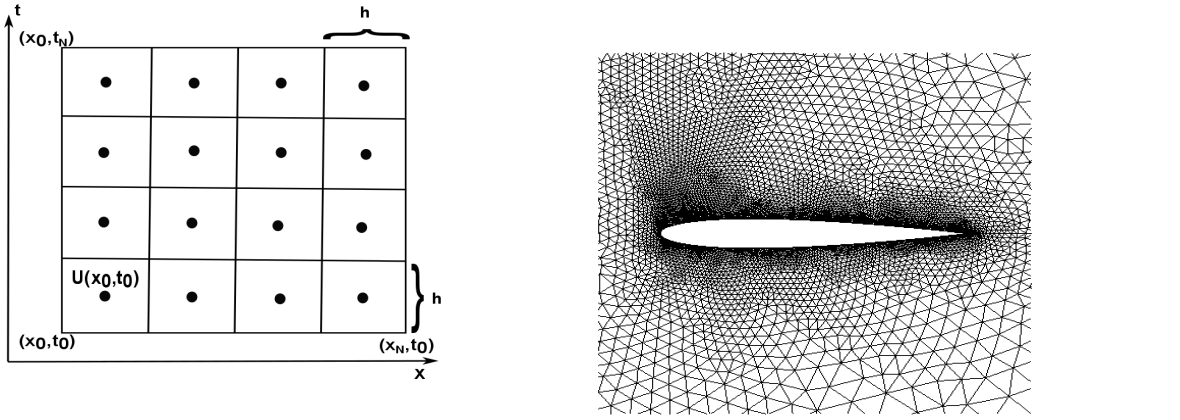

Figure 2.1 **Left:** Uniform grid with rectangular cells containing values for the discrete function $U(x,t)$. The black dots indicate where the value of U is defined for $cell(x,t)$. **Right:** Unstructured grid made of triangular cells around an airfoil[1].

for different values of x and t. In general, the more number of points we use to set up the grid, the better is its ability to correctly cover the physical domain. But generating such a grid can also be computationally expensive and also once generated cannot be modified with much ease.

The example of grid discretisation that we have chosen here shows a grid of uniform width and rectangular cells, with the points that lie next to each other. Given a point in the grid, we can find the neighboring points without much effort. Such a grid is said to be structured due to the regular structure it follows. The cells are typically quadrilateral (2D) or hexahedral (3D) in shape. The regularity of these grids makes memory management simpler which can be exploited in numerical solvers. Structured grids have been mostly used in finite-difference (FD) and FV schemes.

Figure 2.1 (right panel) shows a second type of grid, i.e., unstructured grids. Unlike structured grids, unstructured grids do not have a regular structure due to which connectivity between the points cannot be determined implicitly. Hence, in addition to the location of the points, a connectivity graph that describes the connection between the different points has to be explicitly built. The cells are typically triangular or tetrahedral in shape [96, 125]. Unstructured grids are useful since the cells in such a grid can be placed in a way that they cover the problem domain more accurately, especially near the domain boundaries, and can be used to describe complex geometries but need more effort to maintain. They are mostly used in FE and also in FV analysis [134].

Once a grid has been built, we might want to make local modifications to it by adding or removing points depending on changes in the geometry of the problem at hand. Structured grids although easy to build and maintain are not flexible to such local refinements, unstructured grids are better suited instead. On the other hand, parallelisation of solvers that rely on structured grids are easier to achieve than for unstructured grids. Hybrid grids that combine the advantages of both structured and unstructured grids prove to be a solution in such cases, they are broadly classified into block-structured and hierarchical grids.

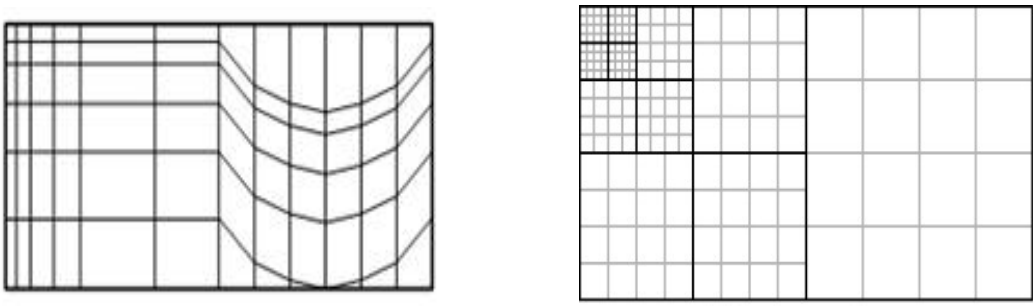

Figure 2.2 **Left:** A block structured grid containing two separate blocks of grid[2]. **Right:** A hierarchical grid, with blocks consisting of cells with different widths.

Figure 2.2, left panel, shows a block-structured grid, which contains multiple blocks each of which is structured (or unstructured), the overall placement of each block in the domain is unstructured. One could, in principle, use only structured blocks here and represent complex geometries as well. These grids are efficient in terms of memory management and solution accuracy since each block can be handled independently. However, special care has to be taken at the interface between two grids. Hierarchical grids on the other hand involve selecting some parts of a grid based on certain criteria, and locally refining it in a structured way. The advantage of such a grid is that the numerical solver within each block can also be structured, which results in efficient performance. Here as well, care has to be taken at the interface between two grids. Figure 2.2, right panel, shows a hierarchical grid with four sets of structured blocks at different cell widths [162].

We now look at the different methods used to generate structured and unstructured grids.

2.1.1 Structured grid generation

There are different techniques used to generate structured grids, the most commonly used methods are the algebraic method and elliptic method. We discuss them in this section. The same techniques can also be used to generate block-structured grids.

2.1.1.1 Algebraic method

Algebraic methods involve transforming the structured computational domain to the physical domain by means of interpolation functions. Once the transformation has been done, the coordinates of the regular domain that have been partitioned into regular intervals map on to the physical domain [186]. The transformation from a computational domain to a physical domain can be written as a vector valued function [170]

$$X(\xi,\eta,\zeta) = \begin{bmatrix} x(\xi,\eta,\zeta) \\ y(\xi,\eta,\zeta) \\ z(\xi,\eta,\zeta) \end{bmatrix} \tag{2.8}$$

[2] http://www.byclb.com/TR/Tutorials/dsp_advanced/ch1_1.htm

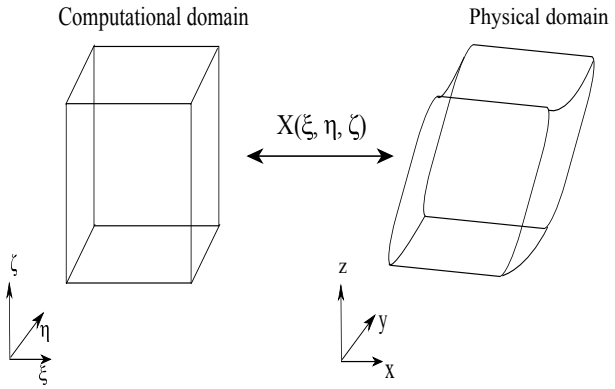

Figure 2.3 Transformation from the computational domain to the physical domain.

where

$$0 \leq \xi \leq 1, 0 \leq \eta \leq 1, 0 \leq \zeta \leq 1 \tag{2.9}$$

Given a set of points (I, J, K) where

$$I = 1, 2, 3, .., \hat{I}, J = 1, 2, 3, .., \hat{J}, K = 1, 2, 3, .., \hat{K} \tag{2.10}$$

the discrete subset of the vector-valued function $X(\xi_I, \eta_J, \zeta_K)$ forms a structured grid where

$$0 \leq \xi_I = \frac{I-1}{\hat{I}-1} \leq 1, 0 \leq \eta_J = \frac{J-1}{\hat{J}-1} \leq 1, 0 \leq \zeta_K = \frac{K-1}{\hat{K}-1} \leq 1 \tag{2.11}$$

The relationship between the coordinates (ξ, η, ζ) and the points (I, J, K) gives a discrete form of the domain with an implicit relationship between the neighboring points. This forms a structured grid. Figure 2.3 shows the correspondence between the computational domain and the physical domain.

Algebraic methods are mainly based on the Transfinite Interpolation method (TFI) which expresses the vector function in 2.8 as a set of three univariate transformations, each of which acts on one coordinate direction. These can then be evaluated in two different ways, the Boolean sum transformation and the Recursion formulation.

2.1.1.1.1 Boolean sum formulation

This method specifies the interpolation in each direction [186], given by

$$
\begin{aligned}
U(\xi,\eta,\zeta) &= \sum_{i=1}^{L}\sum_{n=0}^{P}\alpha_i^n(\xi)\frac{\partial^n X(\xi_i,\eta,\zeta)}{\partial\xi^n} \\
V(\xi,\eta,\zeta) &= \sum_{j=1}^{M}\sum_{m=0}^{Q}\beta_j^m(\eta)\frac{\partial^m X(\xi,\eta_j,\zeta)}{\partial\eta^m} \\
W(\xi,\eta,\zeta) &= \sum_{k=1}^{N}\sum_{l=0}^{R}\gamma_k^l(\zeta)\frac{\partial^l X(\xi,\eta,\zeta_k)}{\partial\zeta^l}
\end{aligned}
\tag{2.12}
$$

and the corresponding tensor products are

$$
\begin{aligned}
UW = WU &= \sum_{i=1}^{L}\sum_{k=1}^{N}\sum_{l=0}^{R}\sum_{n=0}^{P}\alpha_i^n(\xi)\gamma_k^l(\zeta)\frac{\partial^{ln} X(\xi_i,\eta,\zeta_k)}{\partial\zeta^l\partial\xi^n} \\
UV = VU &= \sum_{i=1}^{L}\sum_{j=1}^{M}\sum_{m=0}^{Q}\sum_{n=0}^{P}\alpha_i^n(\xi)\beta_j^m(\eta)\frac{\partial^{nm} X(\xi_i,\eta_j,\zeta)}{\partial\eta^m\partial\xi^n} \\
VW = WV &= \sum_{j=1}^{M}\sum_{k=1}^{N}\sum_{l=0}^{R}\sum_{m=0}^{Q}\beta_j^m(\eta)\gamma_k^l(\zeta)\frac{\partial^{lm} X(\xi,\eta_j,\zeta_k)}{\partial\zeta^l\partial\eta^m} \\
UVW &= \sum_{i=1}^{L}\sum_{j=1}^{M}\sum_{k=1}^{N}\sum_{l=0}^{R}\sum_{m=0}^{Q}\sum_{n=0}^{P}\alpha_i^n(\xi)\beta_j^m(\eta)\gamma_k^l(\zeta)\frac{\partial^{lmn} X(\xi_i,\eta_j,\zeta_k)}{\partial\zeta^l\partial\eta^m\partial\xi^n}
\end{aligned}
\tag{2.13}
$$

where $\alpha_i^n(\xi), \beta_j^m(\eta), \gamma_k^l(\zeta)$ are blending functions. The tensor products are then used to calculate a Boolean sum of the individual interpolations, given by

$$
X(\xi,\eta,\zeta) = U + V + W - UV - UW - VW + UVW \tag{2.14}
$$

2.1.1.1.2 Recursion formulation

The second method of evaluating the interpolation is to use a recursion formula. We begin with the interpolation in one coordinate direction

$$
X_1(\xi,\eta,\zeta) = \sum_{i=1}^{L}\sum_{n=0}^{P}\alpha_i^n(\xi)\frac{\partial^n X(\xi_i,\eta,\zeta)}{\partial\xi^n} \tag{2.15}
$$

and define the second and third interpolations using the first

$$
\begin{aligned}
X_2(\xi,\eta,\zeta) &= X_1(\xi,\eta,\zeta) + \sum_{j=1}^{M}\sum_{m=0}^{Q}\beta_j^m(\eta)\left[\frac{\partial^m X(\xi,\eta_j,\zeta)}{\partial\eta^m} - \frac{\partial^m X_1(\xi,\eta_j,\zeta)}{\partial\eta^m}\right] \\
X_3(\xi,\eta,\zeta) &= X_2(\xi,\eta,\zeta) + \sum_{k=1}^{N}\sum_{l=0}^{R}\gamma_k^l(\zeta)\left[\frac{\partial^l X(\xi,\eta,\zeta_k)}{\partial\zeta^l} - \frac{\partial^l X_2(\xi,\eta,\zeta_k)}{\partial\zeta^l}\right]
\end{aligned}
\tag{2.16}
$$

The TFI method allows control over grid spacing, which can be changed with the use of the blending functions. These determine the number of cells that have to be placed over a certain region of the grid based on specific criteria. Algebraic methods, in general, are easy to implement but suffer from the drawback of producing irregularities in complex geometries.

2.1.1.2 Elliptic method

A second method to generate structured grids is the elliptic method, where a system of differential equations are used to create the grid [162]. Consider, for example, the set of equations

$$
\begin{aligned}
\xi_{xx} + \xi_{yy} &= 0 \\
\eta_{xx} + \eta_{yy} &= 0
\end{aligned}
\tag{2.17}
$$

with specified boundary conditions. We can make use of the maximum principle for elliptic PDEs which states that the maximum values of ξ and η are at the boundary and the interior of the grid has no local extrema. The resulting alignment of grid points defined using the elliptic equation ensures that the grid lines do not crossover. In order to determine the grid points themselves, equation 2.17, which is defined in the physical domain, has to be solved for the computational domain. The following equations result, by separating the dependent and independent variables and applying the chain rule

$$
\begin{aligned}
(x_\eta^2 + y_\eta^2)x_{\xi\xi} - 2(x_\xi x_\eta + y_\xi y_\eta)x_{\xi\eta} + (x_\xi^2 + y_\xi^2)x_{\eta\eta} &= 0 \\
(x_\eta^2 + y_\eta^2)y_{\xi\xi} - 2(x_\xi x_\eta + y_\xi y_\eta)y_{\xi\eta} + (x_\xi^2 + y_\xi^2)y_{\eta\eta} &= 0
\end{aligned}
\tag{2.18}
$$

The above set of equations can be solved using iterative solvers with boundary conditions defined by the boundary curves in the physical domain. Similar to the algebraic method, grid spacing can be controlled by adding source terms to 2.17

$$
\begin{aligned}
\xi_{xx} + \xi_{yy} &= P(\xi,\eta) \\
\eta_{xx} + \eta_{yy} &= Q(\xi,\eta)
\end{aligned}
\tag{2.19}
$$

Elliptic grid methods generate very smooth grids, with an optimal distribution of grid points. However, when compared to algebraic methods, they are computationally more expensive.

We next look at methods to generate unstructured grids and the advantages of each approach.

2.1.2 Unstructured grid generation

There are different methods adopted to generate unstructured grids, the most commonly used among them are the Advancing Front method (AFT), Delaunay triangulation which uses triangular cells and the quadtree or octree methods which are easy to implement but result in large irregularities near the domain boundary. Structured grids can be implemented using simple and regular data structures, for example arrays. Unstructured grids, on the other hand, need more well defined data structures that allow fast searches across the grid. We discuss the AFT method and Delaunay triangulation in this section, we also briefly mention the data structure used to optimise the generating algorithm.

2.1.2.1 AFT method

The AFT technique for triangular grids in 2D was first suggested by George [63], this was extended to 3D [113] and has been modified to work with quadrilateral [23, 139, 205] and hexahedral meshes as well [22]. AFT works by starting with a distribution of points at the boundary of the problem domain, these make up the initial set of edges. Successive steps then involve adding new edges to form cells till the domain is covered completely.

To add a new cell, the following steps are taken.

(a) A list of available edges, L_e, that can be used to form cells is stored. This represents the advancing front which is the separation between the meshed domain and the unmeshed points. A list of points L_p that need to be meshed is also maintained.

(b) Using the points that form the last edge on list L_e, all the unmeshed points that satisfy a certain criteria are selected. For example, the distance between a point and the two points in the last edge should be minimal.

(c) Once such a point has been identified, a triangular cell is created between the point and the edges, the point is removed from L_p. The corresponding edge is removed from the list L_e.

Steps (a) to (c) are repeated till all points have been meshed and no more edges are available to form cells. Figure 2.4 shows the steps taken in the AFT method, the left panel shows the initial front where we only have the points at the domain boundary. The middle panel shows

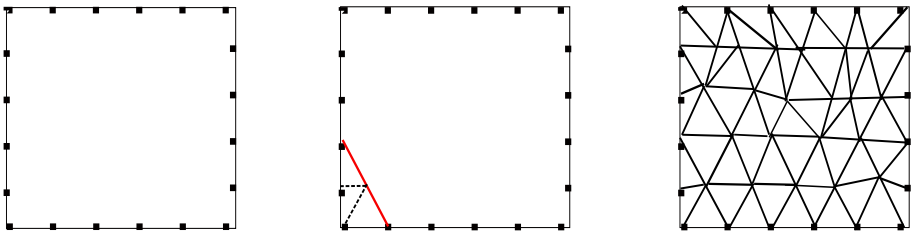

Figure 2.4 AFT method to generate triangular cells, **Left:** Initial front at boundary edges. **Middle:** Advancing the front (red line). **Right:** Final grid.

the front (red line) being advanced by forming edges. The right panel shows the unstructured mesh covering the domain. The algorithm to implement AFT has to check that a valid point has been found before a triangular cell can be created. The new triangle being generated should not intersect with an existing triangle. If the point is not valid, another point from the list L_p can be selected instead. The AFT method can be expensive as it has to perform these checks for each point being added. Its advantage lies in its ability to automatically generate grid points in the interior of the physical domain. Also, since the method starts from the boundary, it ensures that the boundary of the domain has been covered by grid lines.

The AFT method requires a number of sorting and searching operations to build the mesh efficiently. The advancing front needs a data structure to store information pertaining to L_e which requires frequent insert/delete operations. We also need to search the list L_p that contains the next valid point to form the triangle. The Alternating Digital Tree (ADT) [25] has been found to be an efficient solution for both the insert/delete and search operations.

2.1.2.2 Delaunay triangulation

The Delaunay triangulation [54] of a set of points is a dual of the Voronoï diagram [193] which selects and marks a region such that it lies closer to a certain point than any other point. The marked regions form a tessellation of the space and is called a Voronoï tessellation. The "in-circle" criterion then dictates that for any three points in this space, the circle that circumscribes these points does not contain any other point. A unique triangulation can be formed for these points, which is called the Delaunay triangulation. In three dimensions, the same criterion extends to a sphere containing the points which form a tetrahedron instead. Figure 2.5 shows the Voronoï diagram for a set of points and the corresponding Delaunay triangulation (dotted green lines).

There are different methods used to generate a Delaunay triangulation: the Bowyer-Watson method, the Tanemuara-Ogawa-Ogita algorithm and the edge-swapping technique. We discuss all the three here.

The Bowyer-Watson [27, 195] method makes use of the "in-circle" criterion of the Delaunay triangulation. It starts from a very coarse triangulation of the domain. Then, for each point being added, the triangles whose circumcircles contain this point are determined and their

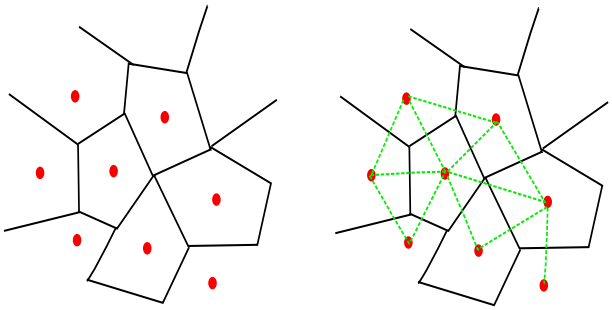

Figure 2.5 Voronoï tessellation of a set of points and the corresponding Delaunay triangulation (dotted green lines).

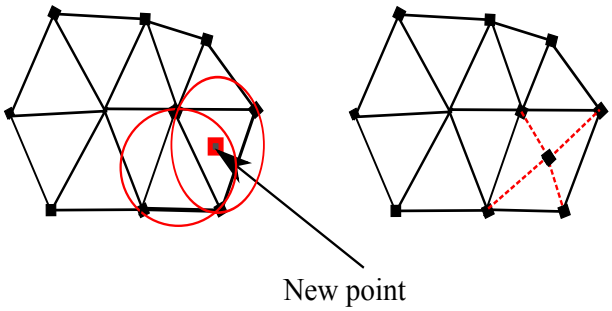

Figure 2.6 Insertion of new point in the Delaunay triangulation using the Bowyer-Watson method. The red lines show the new edges that are formed.

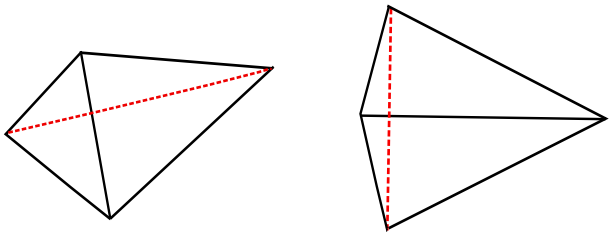

Figure 2.7 Edge swapping between two neighboring triangles, **Left:** max-min **Right:** min-max criterion.

edges deleted. A new triangulation is then formed by creating an edge between the new point and the points that formed the edges of the deleted triangle. Figure 2.6, left panel, shows a sample set of points whose circumcircles are marked in red, the new point being added is also shown. The right panel shows the new edges (dotted red lines) that are formed. This method typically makes use of an octree data structure to keep a list of all the points that have been inserted; this is then searched to find a point nearest to the new point being inserted.

The Tanemuara-Ogawa-Ogita algorithm [183] is similar to the Bowyer-Watson method but starts with a prescribed set of boundary edges and points instead of a coarse triangulation. For an edge e, it constructs a circle C_i that passes through e and another point P_i. The circle that is empty determines the new point P_i that can be used to form a new triangle. If the two edges other than e that form this triangle have not been already defined, then they are defined and added to a list of existing edges. The algorithm stops when all the boundary edges belong to one side of a triangle and each interior point forms a common edge between two triangles. However the usability of this algorithm for a 3D case is rather limited.

A third type of Delaunay triangulation technique is the edge-swapping algorithm where the common edges between two triangles are checked and exchanged if the resulting triangles satisfy certain criteria. This criteria could be the circumsphere test that maximizes the minimum interior angle (max-min criterion) as in [195], or one that minimizes the maximum interior angle [14] (min-max criterion). Figure 2.7 shows such an edge swap being done for two triangles for the max-min (left panel) and the min-max criterion (right panel).

So far, we have looked at generating a grid based on certain initial conditions and the grid structure does not change during the simulation. This method works well as long as we

are only interested in the values of the discretised function $U(x,t)$ (2.4) at a certain cell width h. We cannot resolve further the values of $U(x,t)$ in between the cells due to the grid resolution. This is not ideal as we might be interested in knowing the properties of the fluid in the corresponding sub-domain of the grid under study, for example, the turbulent motion in some regions, interaction between the molecules in the fluid or its thermodynamic properties. Adapting the mesh to resolve certain features in the flow would be important to find the solution in regions with large gradients. AMR schemes provide a stable and efficient solution for such situations and are used in a wide range of scientific applications. We next look at AMR schemes in detail, for both structured and unstructured grids.

2.2 AMR schemes

AMR is a widely used method to discretise and refine the domain, where required, in a complex system. It has been used in areas ranging from astrophysics, global weather modeling and relativity to nuclear fusion modeling and fluid dynamics. AMR makes use of a mesh or a grid to cover the problem domain and refines the mesh wherever necessary.

Depending on the type of grid, structured or unstructured, that is used to cover the domain, AMR can be broadly classified into structured (SAMR) and unstructured AMR (UAMR) [96]. SAMR, when compared to UAMR is better in terms of memory management since the regular structure makes it easier to maintain mesh information. UAMR on the other hand offers the advantage of being able to finely resolve domains that have sharp gradients but needs more complicated data structures to maintain the mesh information.

AMR schemes use a mesh or a grid to describe the physical domain and to progressively increase the grid resolution in certain parts of the mesh based on a set of refinement criteria. By selectively increasing the resolution only in the interesting part of the domain, AMR methods optimize the global memory and computational resource requirements. The adopted refinement criterion in AMR can be, for e.g., a threshold value calculated using the Richardson extrapolation [19], or the weighted sum of the first and second derivatives of some state variables as in [203]. Application dependent refinement criterion have also been used: [198] use the overdensity of baryon and dark matter, [128] adopt a local density criterion while [184] follow the gradients of certain variables describing the gas flow and refine when they exceed a threshold value.

In this Section, we discuss both SAMR and UAMR, citing their pros and cons. We mention some of the applications that make use of both schemes and discuss examples of RT codes as well. There are quite a few freely available AMR libraries based on these methods that are used by the astrophysics community, we discuss them briefly and then focus on one SAMR library, `CHOMBO`, explaining our reasons for choosing this library for the work done in this thesis. The terms mesh and grid will be used interchangeably. Also the term cell and element will be used interchangeably.

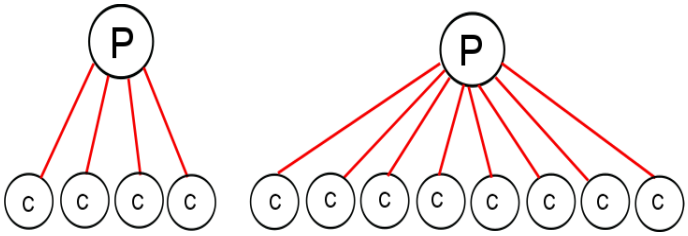

Figure 2.8 Pictorial representation of a quad-tree (left) and an octree (right). The cells with 'P' denote a parent, and those with 'C' denote a child.

2.2.1 Structured AMR

SAMR makes use of rectangular cells to discretize the domain. The cells can be either stored independently or in the form of a grid. SAMR schemes are mostly designed for the solution of systems of PDEs that use FD schemes [24]. Depending on how the cells are stored and refined, SAMR can be classified into three methods, namely cell-based AMR (CBAMR), block-structured AMR (BSAMR) and patch-based AMR (PBAMR). All the three methods are discussed in more detail in this Section.

2.2.1.1 Cell-based AMR (CBAMR)

Cell-based AMR (CBAMR) [204] refines each cell of the grid as and when it is required. The cell being refined is called a "parent" and the refined cells are called the "children" of the parent cell. This sort of refinement is called a "tree-based" AMR due to the hierarchical structure that is formed between the parent and child cells. A tree structure is used to maintain the parent cells, and link them to the respective child cells. The child cells are in turn linked to the parent cell, making up a quad-tree (2D) or octree (3D), as shown in Figure 2.8. A set of cells lying in the same position in the tree hierarchy forms a "level".

In a quad-tree, the parent cell is linked to four child cells, and in an octree the parent cell is linked to eight child cells. The cells at the "root" level or "base" level have no parent cells, and the cells that are not covered by any other cells are called "leaves". Since CBAMR refines on a cell-by-cell basis, it limits the refined regions to wherever necessary (Figure 2.9). In this Section we discuss the CBAMR framework, the description of the grid hierarchy formed by CBAMR, the terminology used to refer to each part of the hierarchy, the time stepping done in CBAMR and the interaction among different refinement levels.

2.2.1.1.1 CBAMR tree description

Figure 2.10, left panel, shows a sample quad-tree structure formed for CBAMR, with a base level and two levels of refinement. The cells have been labeled with the level number and a cell number. The parent cell at Level 0, $C(0,0)$ has links to four child cells, $C(1,0)$, $C(1,1)$, $C(1,2)$ and $C(1,3)$. We use the term "siblings" to refer to child cells of the same parent and "neighbor" to refer to child cells that are spatially adjacent to a child cell. A neighboring

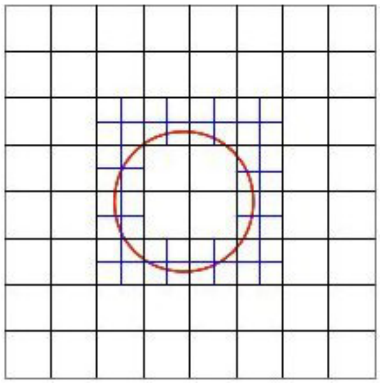

Figure 2.9 Selectively refining cells in CBAMR.

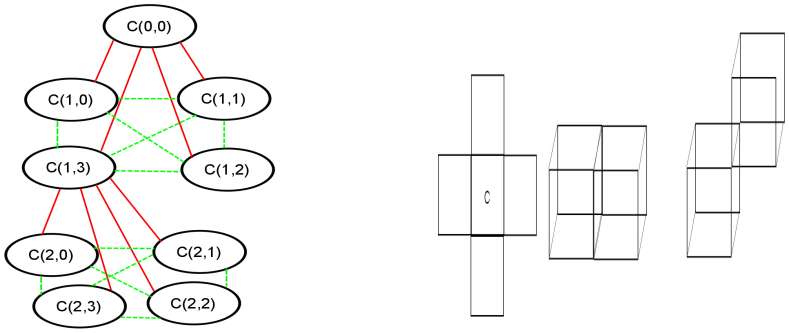

Figure 2.10 **Left:** - Cell hierarchy in CBAMR. Red lines indicate links between parent and child cells. Green dotted lines are for links between sibling cells. **Right:** - Edge neighbors in 2D and face, edge neighbors in 3D.

cell can be at the same level of refinement. If the neighbors are at different levels, then their cell size should not differ by a factor more than the refinement ratio which is the ratio of the spatial widths, h, of the parent and child cells. Hence, in a quad-tree, a child cell will have four neighbors along the edge, while in the octree a child cell can have six face neighbors and 12 edge neighbors [160], as shown in the right panel in Figure 2.10. The links between the siblings in the quad-tree are shown with green dotted lines. Cells $C(1,0)$, $C(1,1)$, $C(1,2)$ and $C(1,3)$ lie at the same level and are siblings. Cell $C(1,3)$ is further refined with four child cells $C(2,0)$, $C(2,1)$, $C(2,2)$ and $C(2,3)$ which form the leaf cells at second level of refinement. Depending on whether a cell is refined or unrefined, these links will have to be updated accordingly.

The tree structure shown here can be expensive to maintain, and requires updating every time a cell is refined or unrefined. One has to maintain a sufficiently large amount of information for a single cell, its parent, child and siblings. Also tree traversal can be complicated if one has to find a neighboring cell that belongs to another parent cell. There have been some optimizations suggested in [96] that allow easier tree traversal to find neighboring cells, and also make CBAMR suitable for parallelisation. The CBAMR implementation, referred to as the "Fully Threaded Tree" (FTT) in [96], groups the child cells into an "oct" which is then

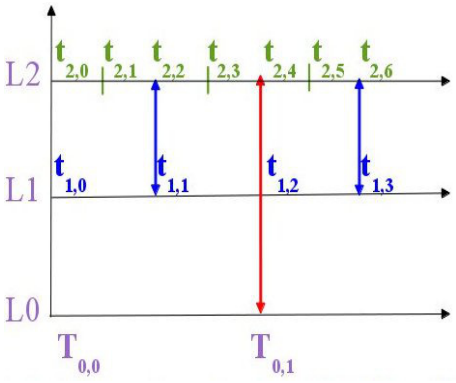

Figure 2.11 Time stepping on different AMR levels, with time step denoted as $T_{l,i}$ at the base level and $t_{l,i}$ at the higher refinement levels, where l refers to the AMR level and i refers to the time step.

used to form the hierarchy among parents and child cells. The child cells of a single parent cell can be stored in memory contiguously, which removes the need for a child cell to have a pointer to its siblings. Also, the child cells can have pointers to the neighbors of the parent cells, this gives information regarding the neighbors that are not siblings but lie at the same refinement level. CBAMR has been implemented in many codes and we will discuss them in some detail in Section 2.2.4.

2.2.1.1.2 CBAMR time stepping and data update

Let's consider the function $U(x,t)$ that we discussed in Section 2.1. At a given time t_0 and a point in space x_0, we can determine the value of the function $U(x,t)$. Now if we advance the time by a small value Δt, we can again calculate the value of $U(x,t+\Delta t)$. This small increment in time Δt is said to be a time step. We can discretize the domain of $U(x,t)$ using the cell hierarchy in CBAMR. Then to refine the domain, the refinement has to be done in space and time since the function $U(x,t)$ changes in both space and time. This essentially means that as the mesh width is reduced on finer levels, the time step on a finer level is also set to be a ratio of the time step on a coarser level. For example, if the width of the mesh for a parent cell is h at Level l, then with a refinement ratio of 2, the width of the child cell at Level $l+1$ is $h/2$. The corresponding time steps for the levels are then Δt and $\Delta t/2$. Figure 2.11 shows an example of time stepping on different refinement levels. The arrows indicate the time step at which all the levels are synchronized before moving on to the next time step, for example when level $L0$ is at time step $T_{0,1}$, all the other levels must be at the same time step, which for level $L1$ is $t_{1,2}$ and for $L2$ is $t_{2,4}$.

We now discuss the time stepping and advancing of the solution in the CBAMR framework on the basis of the FTT implementation discussed in [96]. This solves the Euler equations for an inviscid fluid flow, and the advancing of the solution for these equations for one time step is coupled to the refinement and unrefinement of cells on each level. In order to determine the time step at each level, a global time step is determined using a Courant-Friedrichs-Lewy (CFL) condition which is a necessary condition for convergence while calculating solutions

for a PDE [47]. This essentially requires the time step to be less than a certain minimum value to get numerically correct results. In a 3D scenario, where the octree has leaves whose coordinates are given by three directions, say (u, v, w), the global time step is given by

$$\Delta t_{global} = cfl \frac{2^{-l_{min}} L}{max_i(max_j(a_i + \mid U_{i,j} \mid))} \tag{2.20}$$

where l_{min} and l_{max} are the minimum and maximum level of the leaf cells, a is the speed of sound, $cfl < 1$ is a constant, $U_{i,j}$ is the velocity of the fluid flow and L is the size of the domain. The term a is introduced due to the CFL condition which depends on the Courant number, a dimensionless number given by

$$Courant\, number = v \frac{\Delta t}{h} \tag{2.21}$$

where v is the velocity of the fluid flow, Δt is the time step and h is the width of the cell.

The maximum value of Δt_{global} in 2.20 is calculated for all leaves i in directions $j = (u, v, w)$. The time step on each level is then calculated using the value of Δt_{global} given below

$$\Delta t(l) = 2^{l_{min} - l} \Delta t_{global} \tag{2.22}$$

where l is the level number. Following this, the solution is advanced on each level in each direction (u, v, w), so that we can know the right and left neighbors of a particular cell in each direction. Then the values in each cell are updated using the neighboring cell values.

Having a global time step for the whole hierarchy involves going through all the levels in the tree and doing refinements and advancing at each level. In Figure 2.11, one global time step is completed when we advance from timestep $T_{0,0}$ to $T_{0,1}$ at level $L0$, timesteps $t_{1,0}$ to $t_{1,3}$ on level $L1$ and timesteps $t_{2,0}$ to $t_{2,6}$ on level $L2$. In other words, at the end of a global time step, all the levels must have been advanced to the same point in time. The values in the parent cells are updated using the values in the child cells only after all the levels reach the same coarser level time step. Say, for example, we have two levels of refinement with one level of parent cells and one level of child cells. The data obtained from solving the system of equations, as specified in [96], is maintained in both the parent and child cells. We need to take two times steps on the child cells to reach one time step on the parent cell. It is only after this that the values in the parent cells can be updated by averaging the data from the corresponding child cells.

This concludes our discussion about the different aspects of CBAMR framework. We now look at the technique used in the BSAMR scheme to implement these concepts.

2.2.1.2 Block-structured AMR (BSAMR)

The BSAMR approach is another method of doing AMR and was first suggested in [19]. It can be used to discretise the domain of the function $U(x, t)$ (details are given in Section

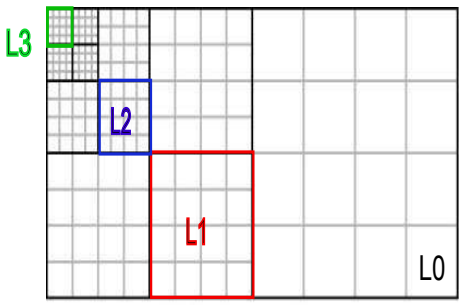

Figure 2.12 Illustration of Block-structured AMR showing the grids at different levels. $L0$ refers to the base grid, $L1$ refers to the grid at one level above and so on.

2.1) over a rectangular grid and the equations for $U(x,t)$ can be solved over the grid. The grid created at the start of a simulation is called a base grid. Then, depending on certain criteria, for example a threshold value in the grid cells, parts of the base grid are refined to a higher resolution. The subgrids at higher resolution are represented as a disjoint union of rectangular grids. A grid on the finer level may overlie more than one grid on the coarser level (see Figure 2.12). Here again we refer to the grid at a coarser level as "parent", the grid at a finer level as "child", and "sibling" or "neighbor" to refer to a "child" grid at the same level. In other words, BSAMR approach also involves a hierarchical structure, but instead of cells, as in CBAMR, the hierarchy is formed using grids at different refinement levels. Similar to CBAMR, BSAMR forms a quad-tree for a 2D grid, and an octree for a 3D grid. In the 3D formulation, this approach is also called an "octree" approach due to the octree formed to link the parent and child grids [208].

In the rest of this Section, we discuss the various components of the BSAMR technique as discussed in [19]. We discuss the terminology used to refer to the grids at each level, the time stepping done at each level, the process of clustering of cells that are selected for refinement, refined grid generation around the selected cells and interaction between grids at different levels. We also discuss the data structures used to store the grid information at each level.

2.2.1.2.1 Grid description

The BSAMR technique makes use of independent block-structured or hierarchical grids, Section 2.1 provides more details on such types of grids. The term 'independent' points to the fact that each grid has separate storage and can be refined or unrefined separately. The system is discretised using a base grid $G0$ whose size remains fixed during the simulation. $G(0)$ itself can consist of multiple grids that are disjoint. Each component of the grid can be referred to as $G(0,j)$, where j is an index of the grid at level 0. The width of each mesh point on the base grid is given by h_0. The base grid can be refined to higher levels that can be further refined. The subgrids at each level are then be referred to as $G(l,j)$, where l is level number and j is the index of the grid. Each subgrid has a mesh width h_l which is an integral ratio of h_0. Each point on the finer grid must lie within the coarser grid unless it is at the physical boundary. Additionally, a finer grid can lie over more than one coarser grid.

An important property of the finer grids in BSAMR is that they are independent from the

coarser grids and are not intermeshed with them. This reduces the amount of complexity required to maintain the data structure containing information about the grid hierarchy and also makes it suitable for parallelisation. The grids at each level can be distributed among the processors depending on the workload associated with each grid. Efficient domain decomposition techniques of SAMR grids is a full fledged research area in itself. The interested reader can find more details in the article by Ralf Deiterding in [149] and references therein.

Now that we have a basic overview of BSAMR grid topology, we look at how time-stepping is implemented for this scheme.

2.2.1.2.2 Time-stepping and Regridding

BSAMR, like CBAMR, also refines the grids both in time and space. For example, if the time step on the $L0$ level is Δt, then on the $L1$ level, that has a refinement ratio of 2, the time step is $\Delta t/2$. So for each time step on the $L0$ level, the solution has to be advanced on $L1$ level by two $\Delta t/2$ time steps to keep both the levels at the same time. Figure 2.11 shows an illustration on how the time stepping at different levels is done. For $L1$ to be at the same time as $L0$, it has to take two time steps t_0 and t_1, for $L2$ this has to be four time steps each of width $\Delta t/4$.

BSAMR grids, like CBAMR grids, adapt over space and time, i.e., as the system evolves over time there might be new regions in the domain that need refinement. Also, regions that were previously refined might no longer satisfy the refinement criteria and hence are no longer required. This can be checked every few time steps to see if the grid satisfies the conditions for refinement. The criteria used to refine a grid can be application specific, for example one could refine a grid based on a threshold value in the grid cells, or on the error estimate of an equation that is solved over the grid data. If the refinement criteria are satisfied, the grids have to undergo regridding, a process that needs to be applied to the grids at all levels.

The regridding process starts from the finest levels down to the coarsest levels. This is done due to the fact that the finest levels hold more accurate data than the coarser levels, and so need to be checked first to see if they need refinement. If we have level l grids, then the finest grid at level l is checked to see if a level $l+1$ grid has to be created. Then the grid at level $l-1$ is checked to see if it needs refinement, and if so it is refined making sure that the newly created level l grid contains the level $l+1$ grid and so on. At each point, we have to ensure that the proper nesting conditions explained in Section 2.2.1.2.1 are satisfied. If a level remains unchanged, then the data for that level is just copied over from the already existing old data.

The process of regridding involves selecting the cells that need to be refined and generating a new grid around these cells. We now look at this method in more detail.

2.2.1.2.3 Clustering and grid generation

One of the main process of regridding involves finding out the points that need to be refined and generating refined grids around these points. To do so, one needs an efficient algorithm

that selects the points to be refined and generates refined grids that cover all the these points whilst ensuring that the size of the refined regions is kept low. We do not want points on the coarser regions that do not need refinement, to be selected while generating the refined grids. The process of clustering, i.e. grouping together tagged cells such that they lie in the same rectangle (if possible), and grid generation is also done when we refine the grid the first time around to create the finer levels.

A method to do efficient clustering of selected points and generating refined grids around these points was developed by [20]. The approach they suggest has been used to do edge detection. The main point of the algorithm includes tagging points that need to be refined and clustering them so that a rectangle covers all these points. To decide where the boundaries of this rectangle lie, the edge detection algorithm is used. The "edge" of the rectangle is taken to be the point where there is transition from tagged regions to non-tagged regions. However, detecting this edge is not a trivial task, as a single non-tagged cell in itself might not indicate a transition. There might be small parts of non-tagged regions that form a kind of "hole" in between tagged regions. The algorithm has to efficiently decide if this is an actual edge and whether to include these points in the rectangle or not.

A major consideration while generating these rectangles is the fill ratio, F_r, a number between 0 and 1, which denotes the efficiency with which the tagged cells are clustered and covered by a rectangle. If the value of F_r is e.g. 0.8, then the rectangles generated are smaller in size, but filled with a higher percentage of tags. This however also increases the number of rectangles generated. If the ratio is low, then the rectangles can be larger in size, but will contain points that do not need refinement. Also, it is necessary that the rectangles generated are disjoint, so that a point does not belong to two rectangles. Figure 2.13 is an illustration of rectangles formed around clustered points. These rectangles will be refined to form the refined grids.

Once the new grids have been generated, data is set on these grids via interactions between the grids at different refinement levels.

2.2.1.2.4 Coarse-fine interactions

The data in the different refinement levels in the AMR hierarchy are coupled to each other, and need to be updated every time a level is changed. This happens in two situations. First, when a fine level is created, the data from the coarse level is interpolated on to the fine level. This is done at the boundary between the coarse and finer levels as well. Second, during advancement of the solution, after all the levels have been advanced upto the same time step, the solution is updated on the coarser grids. The data on the finer grid is averaged on to the coarser levels below. The interpolation of data at the coarse-fine boundaries introduces errors. So the cells that lie on the coarser levels near the coarse-fine boundaries need to be updated using divergence operators to maintain conservation of quantities across levels.

In order to communicate data among neighbors and also while setting data on the finer levels from the coarser levels, ghost cells are used at the grid boundaries. If the ghost cells on the finer levels overlap the data on any of their neighbors, then this data is copied over from the neighbors.

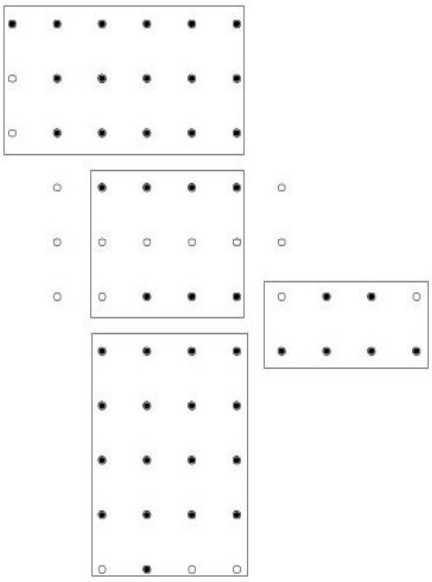

Figure 2.13 An illustration of the clustering of points to form enclosing rectangles [20]. The black dots show the points that are marked for refinement, and the white dots are the points that do not need refinement, but are part of the rectangle.

2.2.1.2.5 Data structures used for BSAMR

Now that we have the whole framework of BSAMR explained, we describe the data structure used in BSAMR to form the hierarchy of grid levels. A tree structure is used to link the grids at different levels. The grid at base level forms the root of the tree and has links to its children. The child nodes in turn have links to their parents and children. A link between child nodes at the same level, or siblings, can also be maintained. Note that this is not a quad-tree or an octree, as typically a grid at a particular level may have many refined grids depending on the refinement criteria and how the refined grid generation has been done, as discussed in Section 2.2.1.2.3. This means that we cannot have a fixed array allocation to point to the child nodes, and need a dynamic allocation method to keep track of the child nodes that are created. This can be done by using a linked list, which can add child nodes or remove them depending on whether a grid is refined or unrefined. Each node in the linked list must contain enough information about the grid it refers to, for example:

- Level at which this grid is located.
- An unique id for the grid.
- Pointers to the parent grids.
- Pointers to any siblings.
- Pointers to any child grids.
- Size of the grid and its start and end coordinates with respect to the parent grid.

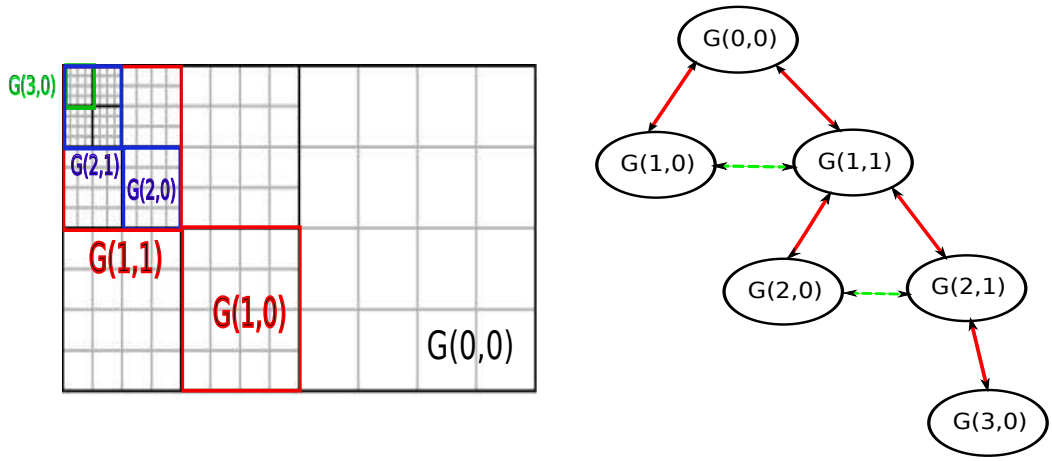

Figure 2.14 **Left:** Grids at different refinement levels in the BSAMR framework. Base grid $G(0,0)$ contains the refined grid at different levels denoted by different colors, Level 1 (red), Level 2 (blue) and Level 3 (green). **Right:** The tree structure created to link these grids. Red arrows link the child and parent grids, the dotted green arrows link the siblings at the same refinement level.

- Array to store the data.

The left panel in Figure 2.14 shows some of the grids at different levels, denoted by different colors. The base grid is denoted by $G(0,0)$, grids $G(1,1), G(1,2)$ are at Level 1 (red boxes), grids $G(2,0), G(2,1)$ are at Level 2 (blue boxes) and grid $G(3,0)$ is at level 3. The right panel in 2.14 shows the tree structure formed to link all these grids. The grids $G(2,0)$ and $G(2,1)$ lie over $G(1,1)$ and are both siblings. Grid $G(2,1)$ is the parent of $G(3,0)$.

We have now looked at the different aspects of the BSAMR framework, we now look at a specific case of BSAMR that has a slightly different method of refining grids.

2.2.1.3 Patch-based AMR (PBAMR)

Patch-based AMR (PBAMR) incorporates the advantages of both CBAMR and BSAMR. In BSAMR, the grid is refined whenever a cell within the grid satisfies the refinement criteria. In CBAMR, individual cells that satisfy the refinement criteria are refined. In PBAMR, the cells that need to be refined are clustered together to form rectangular patches. Hence the patches are placed over the coarser levels only where they are required [50]. The main difference with respect to BSAMR is that the patches at the same level can be different in size and can lie over multiple parent grids. Figure 2.15 shows an example of grids created using the PBAMR framework.

Since a grid at a finer level in the PBAMR framework can lie over many coarser grids, or in other words can have more than one parent, we need to store a list of pointers to all parent grids. The left panel in Figure 2.15 shows the grids at different levels, here a tree structure can no longer be formed since a one-to-one relationship between a child and a parent no longer exists. We can still link the grids at different refinement levels, this is shown in the

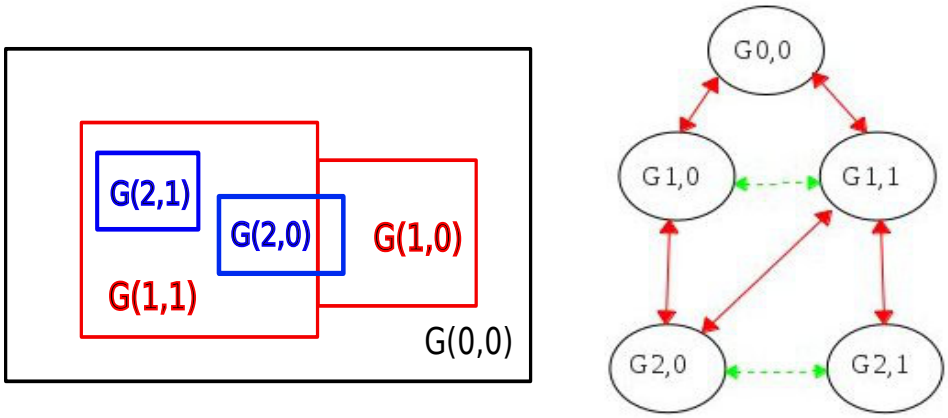

Figure 2.15 **Left:** Grids at different refinement levels in the PBAMR framework. Base grid $G(0,0)$ contains the refined grid at different levels denoted by different colors, Level 1 (red) and Level 2 (blue). **Right:** The hierarchical structure created to link these grids. Red arrows link the child and parent grids, the dotted green arrows link the siblings at the same refinement level.

right panel in Figure 2.15. The grid $G(2,1)$ lies over $G(1,0)$ and $G(1,1)$, i.e., is a child of both grids and so has a pointer to each. In turn both the grids $G(1,0)$ and $G(1,1)$ are the parents of $G(2,1)$ and have a pointer to it.

Similar to the BSAMR framework, the grids at different levels in the PBAMR framework need to satisfy the "proper nesting" requirement. This states that

1. A grid at a finer level must be contained within a coarser grid, that is, the finer grid must start from the end point of a cell in the coarser grid. It cannot lie in between two coarser grid cells.

2. Given a level $l-2$ cell and level l cell, there must be at least one level $l-1$ cell that lies in between these two cells.

Figure 2.16 shows some examples to illustrate the nesting criteria. The left panel at the top shows a grid where the refined cells do not cover the coarser cells completely, this is an invalid configuration. The panel on its right shows a valid scenario where the coarser cells are properly covered. The left panel at the bottom shows a grid with three refinement levels. The box at level $L2$ lies on the boundary between levels $L1$ and $L0$ which is again invalid. The panel on its right shows a valid placement of grids.

The PBAMR framework follows the same approach as BSAMR for its time stepping and regridding operations. At the end of each time step the data on all levels are updated using the coarse-fine interactions mentioned in Section 2.2.1.2.4. The term "patch" and "block" have been used interchangeably in some literature ([35], [4], [154], [184]). Henceforth we will be using the term "patch" and "block" interchangeably to refer to a region of the domain. The refined "patch" or "block" is overlaid on a coarser grid that satisfies the refinement criteria. All "patches" or "blocks" are disjoint and are oriented along the coarser grids.

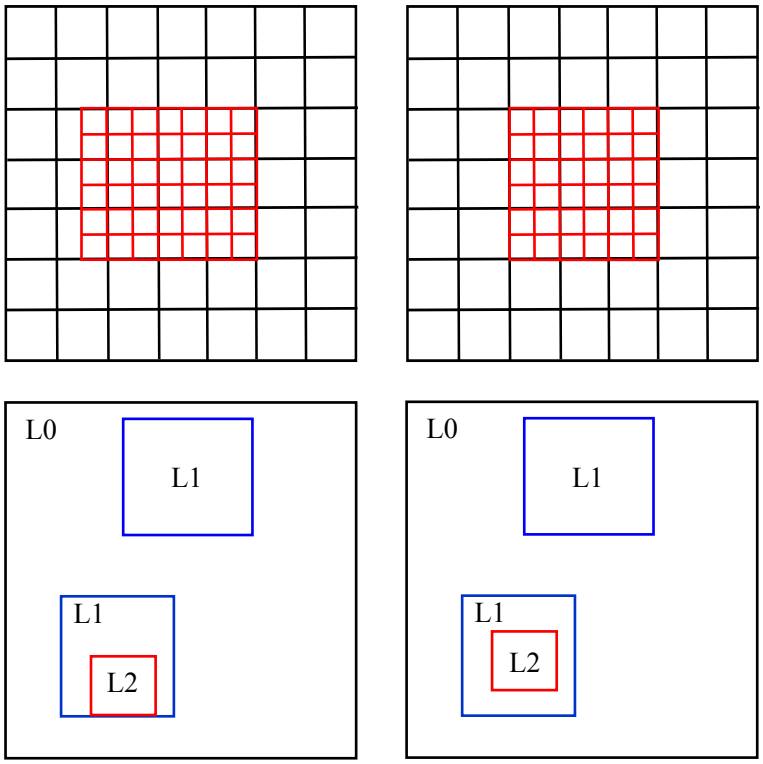

Figure 2.16 Illustration of "proper nesting" criteria of grids. The panels on the left show grid configurations that are invalid. The ones on the right are valid. See text for more details.

To summarise, we have taken a look at the different SAMR methods and the technique used to implement the AMR framework within each method. SAMR methods, in general, offer the advantage of ease of memory management and parallelisation due to their regular structure. However, for more complex geometries, UAMR methods are used. We next look at this framework and the techniques used to implement it.

2.2.2 Unstructured AMR

In Section 2.1.2 we have looked at the different techniques used to generate unstructured grids. We now look at the different methods adopted to introduce AMR schemes into such grids. The motivation to have adaptive methods on unstructured grids is the same as that for structured grids, i.e., to increase the resolution in the areas of interest so that parts of the domain that exhibit rapid changes can be represented with higher resolution. This improves the efficiency of the code, at the cost of increasing its complexity. However, the gain in efficiency more than offsets the effort required to develop adaptive unstructured mesh codes.

Mesh adaptivity for unstructured grids can be broadly classified into three types depending on how the existing mesh is changed. These include

1. The 'r'-refinement, which refines the locations of the nodes; this does not change the total number of mesh points, but instead clusters the cells in the regions of interest.

The connectivity information of the cells is also not changed.

2. The 'h'-refinement, which refines a "coarser" cell into multiple "refined" cells; the overall topology of the mesh does not change as the refined cells cover the space occupied by the coarser cell. This however increases the total number of cells in the mesh. This method can be further classified into Isotropic and Anisotropic refinement. Isotropic methods refine a D dimensional mesh in all dimensions, for example a 2D mesh will be refined in both the (x, y) directions. Anisotropic methods instead are used when the mesh needs to be refined in one particular direction, for example, a flow feature which is limited to one direction.

3. The 'p'-refinement, which increases the resolution by increasing the order of the polynomial approximation within each cell. This does not change the total number of cells in the mesh or the mesh itself.

4. The 'hp'-refinement method, that combines both the 'h' and 'p'-refinement methods.

The 'h'-refinement method falls under the category of UAMR. It has been found to be suitable in parallel applications and is widely used. Hence we look at the different aspects of UAMR schemes required to generate the new mesh in addition to the existing mesh. These include error estimation to decide which cells need to be refined and mark them for refinement. The mesh itself is then modified using different techniques which will be discussed as well.

2.2.2.1 Error estimation

UAMR methods are mostly used for FE and FV methods. Here we briefly discuss the error estimation methods for both cases. Initial error estimators for FE based methods were *a priori* or interpolation methods [55], but these were not found to be accurate in situations where the domain had complex features, for example, boundary layers. The need for an accurate estimation of the error led to the development of *a-posteriori* error estimation methods [3], which use the computed solution to check if the mesh needs to be refined further.

The important factor in error estimation is the error estimator η_τ for each cell (or element) T_τ in the mesh. This depends on the solution calculated in T_τ and its neighboring elements in addition to the boundary conditions specified for the problem [7]. Then

$$E_Z = \left\{ \sum_{\tau \in T_\tau, \tau \subseteq Z} \eta_\tau^2 \right\}^{1/2} \tag{2.23}$$

where E_Z is the estimate of the error in a subdomain Z. If $e_h = u_{EX} - u_h$ where u_{EX} and u_h are the exact solution and the FE solution of the problem respectively, then the aim is to satisfy the criteria below

$$\widetilde{C}_L E_Z \leq |||e_h|||_Z \leq \widetilde{C}_U E_Z \tag{2.24}$$

where $|||e_h|||_Z$ is the norm of the error in the subdomain Z and $\widetilde{C}_L$, $\widetilde{C}_U$ are constants with values close to 1. To allow for the estimator to be used in AMR, 2.24 has to be valid in small subdomains as well.

A good error estimator must necessarily have a computational cost smaller than the cost of computing the solution, and should be able to find an error estimate for different mesh spacings. There are different types of error estimators, which can be broadly categorised as

(a) Residual error estimators that depend on the solution of a local problem that is analoguous to the higher-order FE approximation of the original problem. These are again categorised into implicit or explicit estimators [3]. This has also been used in the FV method [90].

(b) Flux-projection error estimators that calculate the difference in the flux between the original FE estimate and that obtained by post-processing the FE solution [207].

(c) Extrapolation error estimates that compare two FE solutions on different meshes to get an error estimate [182].

(d) Interpolation error estimates [56].

a-posteriori error estimation methods for FV schemes are mostly based on residual error estimates, however goal-oriented error estimation schemes that estimate the errors only for variables of interest are also being used [37].

Once the error estimator has been applied, the elements that need to be modified can be identified and the mesh can be modified accordingly.

2.2.2.2 Modifying the existing mesh

The procedure to modify an existing mesh to generate new ones depends on the error in the calculated solution as computed by the error estimator. Any refinement done to the mesh must result in a *conforming* mesh. In 2D terms this essentially means that the intersection of any two triangles should either be a line segment connecting two points, a single point or an empty set.

One method to divide a triangle is to subdivide it by determining the centroid of the triangle and using it to bisect the edges of the triangle, this approach can result in a nonconforming triangle and also the angle between the edges can approach 0 or π. This can degrade the quality of the FE method [93].

Another method that is used is the bisection method, which divides a triangle in half to generate two triangles. Once an element has been bisected, the resulting mesh is checked to see if it is conforming. If not, then the nonconforming elements are also bisected till the final mesh is conforming [93]. A third method, called regular refinement, suggested by [12] divides the triangle into four smaller elements by joining the centroid with all edges. The triangles that share a bisected edge are marked as "green triangles" and are divided as well. Figure 2.17 shows the three methods being applied to two triangles. In each case, the resulting triangle is nonconforming.

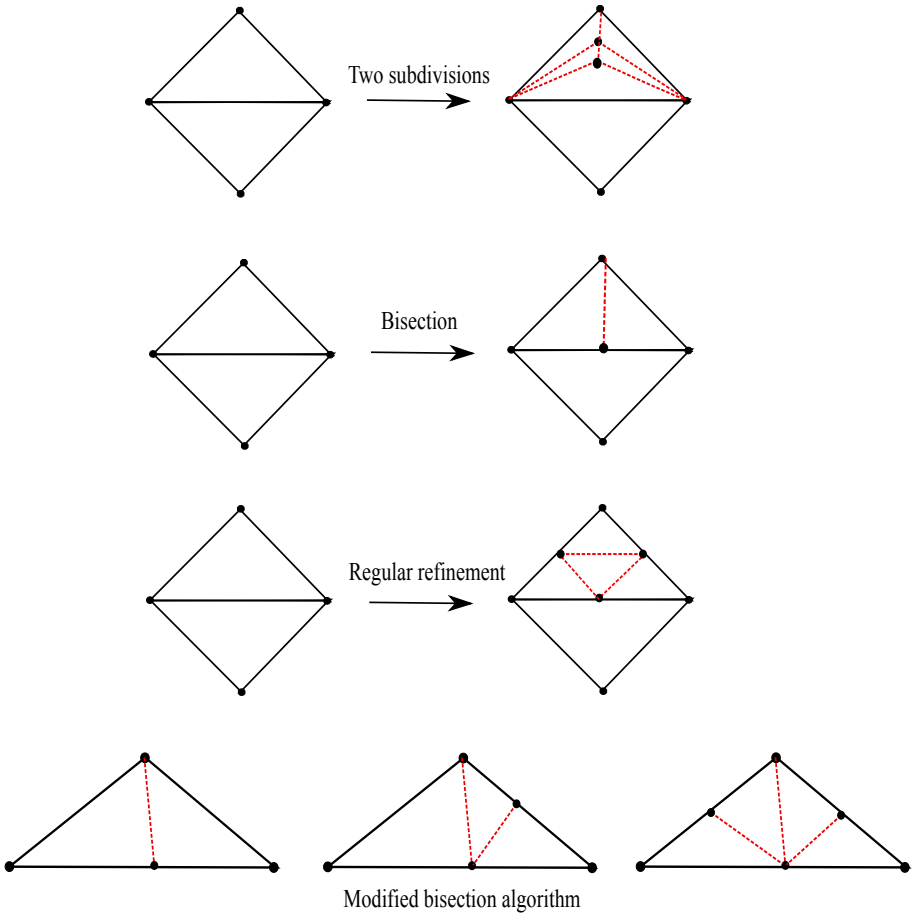

Figure 2.17 Different schemes to divide triangles into multiple elements. **Top:** Two subdivisions along the triangle centroid [93]. **Second from top:** Triangle bisection that divides a triangle into two exact halves [93]. **Third from top:** Regular refinement of triangles, using green edges. **Bottom:** Mesh refinement based on a modified bisection algorithm [156].

In [156], another mesh refinement technique was suggested that bisects a triangle based on its longest edge. It marks the elements to be refined using error estimators and refines them. Any other nonconforming triangles are also marked for refinement. This is done till the mesh is conforming. The number of iterations to generate a conforming mesh, using this technique, is finite and the angles in the resulting mesh elements lie within the range $(0, \pi)$. A variation of this technique first bisects the triangle along the longest edge, and any nonconforming edges are bisected again even if they are not the longest. Figure 2.17 (last panel) shows this modified version of the algorithm; the first bisection is along the longest edge and subsequent bisections might not follow this rule. An alternative to this algorithm was suggested by [78] where the triangle is bisected along the edge that has the maximal value of its length multiplied by a "mesh density function" defined, a priori, at the middle of the edge.

In the case of 3D grids, conformity of the mesh implies an additional constraint that the intersection of two tetrahedra is a triangle. An extension of the algorithm suggested in [156] was suggested by [158] for the specific case of tetrahedra. However, it was not clear if the quality of the mesh degraded over time due to the refinement and also if a finite number

of similar tetrahedron are generated. The methods suggested by [9] and [112] also generate tetrahedra similar to the one in [158] and provide proof that the resulting elements are similar and finite in number.

As a final step to mesh adaptation, we mention coarsening of the mesh which is useful in time-dependent computations where we unrefine a region that was refined in the previous step. This has to be done carefully in the case of triangles and tetrahedra since they cannot be simply merged and form a conforming mesh. A method suggested in [157] unrefines the mesh elements using the bisection method.

Now that we have given an overview of SAMR and UAMR methods, lets look at the advantages and disadvantages of each method.

2.2.3 Pros and cons of SAMR and UAMR

From our discussion above, we know that SAMR and UAMR methods are suited to certain problems and have their own advantages in terms of efficiency, accuracy of solution and effort required to develop such a code. In the absence of any adaptivity, unstructured meshes require more complex data structures than structured grids, and the numerical methods for solving the equations on such meshes are also more complex. When adaptivity is taken into account, both meshing schemes have equal complexity. UAMR methods provide an advantage of being able to generate meshes that match the specific discretisation of the problem. Also the meshes generated can align with the geometric features of the problem being looked at [110].

The regularity of grids in the SAMR approach make it possible to develop fast and efficient computational kernels based on FE and FV methods. This is not the case with UAMR due to the irregular connectivity among the different grid elements. SAMR schemes also have relatively less memory requirements due to the implicit connectivity among the grid elements. In contrast, UAMR schemes need to maintain an explicit connectivity graph thus consuming more memory.

The presence of grid blocks increases the level of granularity in SAMR, parallelisation of this method hence requires a proper load balancing scheme to ensure that the grids are distributed evenly among the cores. Repartitioning the entire grid is not required in SAMR, only the grids created in the new level need to be again distributed among the cores. Load balancing for UAMR schemes is relatively easier as the grid is partitioned using individual elements. However, the repartitioning of data can result in large scale migration of data among cores and requires good strategies to ensure data locality [81].

SAMR schemes maintain numerical accuracy by use of interpolation schemes and interactions among the grid levels which make it harder to employ higher-order schemes. UAMR methods on the other hand do not require any such interpolation schemes since the conformity of the mesh is ensure throughout [33]. Unlike SAMR methods, UAMR methods make use of mesh generation and mesh insertion schemes, as discussed in Section 2.1.2 to introduce new elements. This requires additional checks to ensure that the mesh quality is preserved [81].

This concludes our discussion about the different types of AMR schemes that are employed. We now take a look at some of the applications of AMR, focusing on examples from the astrophysics community.

2.2.4 AMR based applications

AMR has been widely used in many scientific applications such as numerical relativity, global weather modelling and nuclear fusion modelling [149]. The field of astrophysics is particularly well suited for the use of AMR due to the large dynamic ranges involved. Examples range from simulating a coronal mass ejection from the solar surface [75], the expanding front of a supernova explosion [64], gravitational collapse of cold, dense gas to form stars [99], reionisation from Pop III stars [198], to studying the effect of reionisation on star formation in low-mass dwarf galaxy haloes [169].

A large number of simulations have been done to understand the formation of the first structures and the subsequent galaxy formation and evolution [see 103, and references therein]. To simulate collisionless dark-matter particles N-body methods are used [6, 28, 79, 97, 166, 188], while hydro simulations follow the gas dynamics resulting in formation and evolution of objects like galaxies and stars [164, 171]. RT simulations to study radiative feedback effects are usually used as post-processing tools for hydro codes or N-body codes; they form a major aspect of the study of structure formation in general, and the IGM reionisation in particular (Gnedin and Abel 69, Iliev et al. 87; see also Ciardi 38 and references therein). Such large scale hydro and RT simulations require a huge amount of computational and memory resources to be able to handle spatial resolutions that span several orders of magnitude. This problem has been resolved to a great extent by the use of AMR schemes [see 31, for a discussion].

The Adaptive Refinement Tree (ART) code by Kravtsov et al. [102] is an N-body code used for cosmological simulations. The code has been used to study the structure of dark matter haloes with higher resolution. The code makes use of the CBAMR framework to do AMR. The tree structure used to link the parent and child cells is similar to the approach in FTT where the child cells are grouped in a block called "oct". A doubly linked list is used to link the parent and child cells, and refinement and unrefinement of cells can be done to adjust the tree structure.

Many hydro codes make use of AMR schemes to carry out gas dynamic simulations over a range of spatial scales. Some of them are also coupled with RT schemes to perform self-consistent simulations where the RT feedback is accounted for in the dynamical evolution of the gas. The Hydrodynamics Adaptive Refinement Tree (`HART`) code [71, 102] uses the OTVET approximation for the 3D RT implementation [69], while the cosmological hydrodynamics code `RAMSES` [159], designed for simulations of structure formation, incorporates RT using the M1 closure formalism [109]. Both `HART` and `RAMSES` make use of the FTT described in [96] to implement AMR. `RAMSES` makes use of the CBAMR framework since the domain has a very complex geometry because of the hierarchical structure formed due to halo clumping. The child and parent cells are linked to each other with a doubly linked list which makes it more efficient to modify the tree. One notable difference between the FTT implementation in [96] and [102] and the approach in `RAMSES` is that no unrefinement criteria are applied to the cells in `RAMSES`.

The `RAGE` radiation hydrodynamics code [68], makes use of the CBAMR method to implement AMR. It makes use of an octree structure to refine the parent cell into eight child cells. Cells can be refined and unrefined according to the refinement criteria, which adds or removes child cells from the cell hierarchy. The load balancing is done by distributing the cells among the

processors, by using space filling curves to determine the cell to processor allocation. RAGE has been used in models that study the impact of asteriods on a Martian surface [148], and also to study the interaction of jets with clumpy media where the underlying structure being studied is similar to that found in jets from young stars [197].

As a final example (see also [149] for more examples) we mention ENZO [30, 198], a well known code in the cosmological community which uses hydro and N-body schemes coupled with an adaptive ray-tracing scheme implemented in the HEALPix library [1, 72]. The interested reader can find more examples in [88].

AMR can be important also in Magneto Hydrodynamics (MHD), for example in CHARM [128] AMR is necessary to include gravity and collisionless particle dynamics, while in FLASH [62] it is used to make detailed studies of thermonuclear flashes. The PLUTO code [127], is a high resolution, multi-physics code used for solving equations of MHD and has been used to study accretion disks and stellar and extragalactic jets.

Enabling AMR in a stand-alone RT code makes it suitable to post-process the output of many grid-based hydro codes that use the same AMR logic. By representing the regions of interest with high resolution grids, we can follow the growth of the H II regions around the ionising sources and understand how RT feedback effects contribute towards structure formation in grid-based hydro codes. A number of stand-alone RT codes implementing AMR exist: some examples are RADAMESH [35], which is an MCRT code with a ray-tracing scheme. It uses AMR to accurately represent the Ionisation fronts (I-fronts) propagating away from the sources. FTTE [153] is another example that implements a scheme to do RT on refined grids in the presence of diffuse and point sources, and finally IFT [5] which has been developed to explicitly follow the I-front around a point source. [116] have developed an MC RT code for carrying out simulations on the AMR grids generated by the ENZO.

All the examples mentioned above make use of the SAMR framework. In general, RT codes are used as post-processing tools for hydro codes. The trend till recently has been to use structured grids along with either the SPH method [150] or SAMR based methods to do adaptive hydro simulations. As a result RT codes have also adopted the same methodology. However, recently some hydro codes have made use of UAMR as well for being able to resolve regions with more accuracy when compared to SAMR. The AREPO code makes use of a moving unstructured mesh created using Voronoï tessellations [172]. TESS [58] is another MHD code that uses unstructured grids to solve the equations of compressible hydrodynamics for relativistic and non-relativistic fluids. A few RT codes that make use of unstructured grids also exist; [140, 141] code is an RT code developed for use with dynamic, unstructured grids. SKIRT [10, 34] is another code that also makes use of Voronoï tessellations to do RT on unstructured grids, but is used as a post-processing tool and does not implement adaptivity within the RT. Similarly, the LIME code [29] is another example that uses unstructured grids and has been used for modeling ALMA [3] data.

Our discussion so far makes it evident that AMR methods are in wide use in the astrophysics community and will continue to be so. With the availability of increased computing power, the aim is to be able to carry out simulations that take into account the full dynamic range in space and time required to understand the theory of large scale structure formation. AMR

[3] http://www.almaobservatory.org/

is an important tool to achieve this goal. There are a number of computational groups that implement the AMR framework in freely available libraries, this allows the users to quickly introduce AMR into their codes without much effort. We now look at some of these libraries and their applications.

2.2.5 AMR libraries

There are a number of free AMR libraries available based on the different techniques/frameworks discussed above. Each has its own advantage due to the framework it is based on. Many codes that use AMR have their own AMR implementations as this makes it easier to tailor the implementation to the specific application and makes a code easily manageable. Also, support for libraries from developers is discontinued sometimes, which could in principle make any library related issues difficult to resolve. Nevertheless, AMR libraries are widely used as they save a lot in terms of development time, and are done by experts in this field. We briefly discuss the libraries below, the framework they are based on, and mention some of the applications that use them. The list is by no means exhaustive, and only serves to give a general idea of the trend that is followed in the design and development of such libraries.

`CHOMBO` [2] and the `SAMRAI` [4] library [199] are freely available AMR libraries, written in C++, which are based on the BSAMR framework. The libraries are organised into classes, each of which provides a specific function necessary for doing AMR. This makes it easier for a user to incorporate AMR into their application as they need to only focus on implementing the physics. The grid generation and management, refinement and time-stepping are taken care by the library. Also, output files can be written out in the HDF5[5] format that can be visualised using Visit [6]. Both libraries are also useful for MPI-parallelised applications.

A salient feature of `CHOMBO` is that it makes use of Fortran in the computation of intensive parts of the library where data stored in multi-dimensional arrays need to be used. It also takes care of the interfacing required between C++ and Fortran. `CHOMBO` makes use of macros, that depending on the dimensionality required for the application, ranging from 1D to 6D, can tailor the library code. This makes the code essentially dimension independent. The user while configuring `CHOMBO` can select the dimension required for the application and does not need to do any other changes.

The `PARAMESH` [7] library by [119], which is freely available under the NASA-wide Open-Source software license, is also based on the BSAMR framework. `PARAMESH` is written as a set of Fortran90 routines and also provides a C interface. The library manages the grid structure by forming an octree where the parent grids have a link to the child grids and their neighbors. The grid that satisfies a refinement criteria undergoes refinement repetitively, forming child grids, until a certain resolution is reached. This library is also suitable for MPI-parallelised applications.

The `BEARCLAW` package [49], which is an extension of the `AMRCLAW` package [18], written in Fortran90, also makes use of the BSAMR framework and implements a tree structure to maintain

[4] `https://computation.llnl.gov/casc/SAMRAI/SAMRAI_Software.html`
[5] `TheHDFGroup,http://www.hdfgroup.org/HDF5`
[6] `https://wci.llnl.gov/codes/visit/home.html`
[7] `http://www.physics.drexel.edu/~{}olson/paramesh-doc/Users_manual/amr.html`

the hierarchy of grids at different levels. AMROC [53] is another library that implements the BSAMR framework and is written in C++.

There are also libraries that implement AMR on unstructured meshes, PYRAMID is a freely available library [114, 136] that can be used to generate unstructured grids for use with parallel FE and FV models. The library makes use of ParMetis [95] to partition the unstructured grids across the computer cores. SUMAA3D [60] is an MPI based library that provides routines that handle mesh generation, AMR and mesh optimisation. The library is interfaced with the solver package PETSc [11]. libMesh [98] is another library for solving PDEs using unstructured meshes on serial and parallel platforms.

We would like to highlight the fact that the codes CHARM and PLUTO make use of the CHOMBO library to implement AMR. The FLASH code was, until a few years back, using the PARAMESH library to implement AMR. However, PARAMESH is no longer supported and support for any bug fixes is no longer provided. Hence, developments to enable FLASH to make use of the CHOMBO library are underway. Another point to be noted is that FLASH is written in Fortran and CHOMBO is a C++ library, which demonstrates that the library is interfaceable with other codes. Hence, we will discuss the CHOMBO library in more detail.

2.2.6 CHOMBO Library

CHOMBO is a freely available AMR library; it is being actively developed at Lawrence Berkeley National Laboratory (LBNL) and implements a PBAMR scheme on a C++ framework to solve systems of hyperbolic, parabolic and elliptic partial differential equations; CHOMBO has been successfully used by many gas dynamic codes such as CHARM [128], FLASH [62] and PLUTO [127].

Before an application uses AMR, it is important to understand the reason why AMR is being used in a particular code, and whether it provides any advantages. Once it is accepted that AMR is required, a library can be selected which should be easy to use, and also can be used in parallel applications, to provides any benefits of AMR. We discuss some of the performance studies that have been done for CHOMBO with respect to its efficiency and scalability in a parallel application, in Section 2.2.6.1.

CHOMBO is organised into a hierarchy of classes each of which provide a specific functionality. In subsection 2.2.6.2 we discuss these classes in more detail, along with the terminology used in the library to describe the BSAMR method by [19].

2.2.6.1 CHOMBO feasibility

In this Section we discuss some of the feasibility studies that have been done for CHOMBO.

There have been studies done with the CHOMBO library on the Cray XT4 and XT3 systems to determine its scaling efficiency on a large number of cores [189]. The benchmark code chosen was an explicit method for unsteady gas dynamics in 3D and CHOMBO optimisations were applied to this benchmark code. The library developers opted to do benchmarking based

on "replication scaling" where a grid hierarchy and data that was set on a certain number of cores, is replicated over all the cores on which the code is being benchmarked. In other words, multiple copies of the same simulation are run on multiple cores.

The benchmark code used AMR to only refine the grids once. There was no time dependent grid refinement or "regridding" done, which is usually the case with the BSAMR approach. Also the time taken to setup the grids initially were not taken into account for the benchmark.

The optimisations done to CHOMBO focused on the load balance among cores and the inter-core communication for grid metadata information. To optimise load balancing, the developers used Morton ordering [131] to split the domain among cores. This was found to be better than the algorithm based on the Kernighan-Lin algorithm [111] used earlier.

The next optimisation was done to the grid metadata contained in each core. Each core needs to have the information about the neighbors of the grids it contains. This global metadata information is stored as a vector of tuples, where each tuple contains the box id and the core id containing that box. This information is necessary while copying data at the grid boundaries where ghost cells are placed to store boundary data. Using this metadata to get the grid information was found to be inefficient while scaling to a large number of cores. So optimisations that included caching the metadata were done to the library.

With the above mentioned optimisations, the benchmark code was run on cores ranging from 128 to 8192. The simulations consisted of three AMR levels, one base level and two refinement levels, with a refinement ratio of four among each level. The domain size was 16^3 on the coarsest level, and the run was done for one coarse level time step, which meant 16 time steps on the finest level. It was found that the parallel efficiency achieved on all the runs was around 96% on the Cray machines. This is very hard to achieve and proves that CHOMBO is well suited for parallelisation on a large number of cores.

In another optimisation exercise undertaken by the developers, the metadata storage was further improved [190]. In the original approach, the metadata pertaining to the grids increased in size as the number of cores and number of grids in the simulation increased. This was changed to use a bitmap that stored a 1 if a grid belonged to a core and 0 if not. This bitmap was then compressed and stored. This optimisation was found to improve the memory performance quite substantially even on 196K cores.

From the above studies, it is evident that the library scales efficiently on a large number of cores. This is an important requirement for an AMR library which ensures that applications using it can be scaled on a large number of cores. Next, we look at the PBAMR framework in CHOMBO to get an overview of how it is implemented in the library.

2.2.6.2 PBAMR framework in CHOMBO

CHOMBO is based on the PBAMR framework and discretizes the problem domain in rectangular grids. The discretization can be cell-centered, face-centered or vertex-centered as shown in Figure 2.18. The grids generated by CHOMBO satisfy the "proper nesting" requirement (Section 2.2.1.2 provides more details about this criteria).

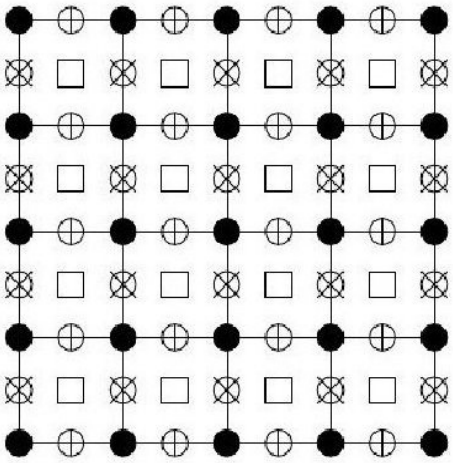

Figure 2.18 Vertex •, Cell □ and face ⊗, ⊕) centered points on a grid, CHOMBO design document [2]

We need to define some terms here, to refer to the data that is stored in the grids. From a numerical point of view, the equations that are solved on a grid at a certain level are done only on those regions that are not covered by finer grids. Such a region is said to be "valid". Any part of a grid that is covered by finer grids is said to be "invalid". This indicates that the data in the "invalid" region does not represent the exact solution to the equations being solved, and that the overlaying finer grid should be used. Additionally, a "composite array" is the collection of valid regions for each level of refinement [128].

The library is organised into hierarchy of classes each of which provides a specific functionality for incorporating AMR into a stand-alone code with minimum effort, so that the code developers only need to focus on implementing the physics. Some necessary but routine tasks associated for example with grid generation and management, its refinement and time-stepping, are automatically managed by the library. Hereafter we describe some of the functionalities implemented by CHOMBO and the classes associated with these functionalities.

- **In-memory data organisation and grid processing**. The *IntVect* class represents points on the rectangular lattice Z^D where D is the dimension of the grid, which in our case is 3. The *IntVectSet* class holds a set of IntVect points to represent the coordinates of the grid. The *Box* class represents a rectangular grid in Z^D and can be coarsened or refined. At each refinement level, the grid is represented as an array of disjoint boxes (*DisjointBoxLayout*) which can be traversed with iterator classes (e.g. the *DataIterator* and *LayoutIterator* classes) to access single object instances. Boxes that are neighbours of some other specific box can be accessed instead by the *NeighborIterator*.

  To store the data in each grid cell we used the Fortran Array Box class (*FArrayBox*), which uses Fortran arrays. The data in *FArrayBox* can be viewed as a $D + 1$ array. The extra dimension is added to indicate the different physical variables or components that can be stored in the D dimensional array. For example we can store the gas temperature or particle velocity in the *FArrayBox* and they form two components. The user can either refer to each component separately or together as a set and carry

out arithmetic operations on them. The class also offers the advantage of providing a Fortran pointer to each component stored in the grid and speeds up operations of data access and update. The *Interval* class refers to the complete set of components that are stored in the grid.

- **AMR hierarchy management**. `CHOMBO` provides the *AMR* class to handle the different refinement levels of the *SAMR* framework. Each refinement level in the hierarchy is represented by the *AMRLevel* class with pointers to the relevant parent or child level for easy traversal.

- **Interactions between different refinement levels**. To maintain continuous gradients of the physical quantities stored across different levels, AMR schemes adopt interpolation and averaging methods at the interface of grids. To initialize the fine grid from the existing coarse grids the *FineInterp* class can be used, while to update the coarse grids with the data on the finer grids the *CoarseAverage* class is adopted. Hereafter the term "refinement ratio" will define the ratio of the spatial widths between the cells in two adjacent levels of the AMR hierarchy.

- **Data storage and grid data I/O**. `CHOMBO` uses the HDF5 data format standard for I/O of data, this can be easily post-processed and visualised by using state-of-the-art visualisation software like Visit or Paraview which have `CHOMBO` plugins available.

2.3 Summary

In this Chapter we have looked at the different methods used to generate structured and unstructured grids and the advantages each type provides. We then looked at the different AMR schemes that are available to carry out adaptive refinement within grids. These are important to model the evolution of a physical system over time, and also to focus computational resources where required. We discussed a few AMR libraries and their use in the astrophysics community. We also looked at a few examples of hydro and RT codes that make use of AMR. It is clear that RT codes stand to benefit from using AMR schemes in order to do high resolution simulations. So, we have discussed in some detail the different AMR libraries that are available and then focused on one open source AMR library, `CHOMBO`, highlighting its salient features. Next, we will take a look at a RT code called `CRASH` and the techniques for using `CHOMBO` library with this code to carry out efficient RT simulations.

Chapter 3

Radiative Transfer on static, nested grids in CRASH

In Chapter 1 we have discussed the role of RT feedback effects on the IGM reionisation and large scale structure formation in the universe. In Chapter 2 we mentioned a number of examples of hydrodynamic codes coupled to RT codes and also stand-alone RT codes that are used in post-processing. One such stand-alone RT code used in post-processing with hydro codes is CRASH, it adopts the MC scheme to do RT simulations. CRASH has been applied to a number of scenarios, for example to study H and He re-ionisation [41, 42, 43] and the role of RT effects on a fluctuating UV background [122].

From the previous chapters, we know that AMR is a widely used methodology in the scientific community to focus computational resources where needed. We have also looked at a large number of examples from the astrophysical community that make use of AMR. It is clear from the discussion that hydrodynamic codes and stand-alone RT codes stand to benefit from the advantages that AMR provides. This incentivises us to develop CRASH so that it can also gain from the AMR technique. SAMR is the preferred choice of most RT codes since they can then be easily coupled to a hydro code, most of which also use SAMR. Also, many applications make use of the BSAMR method for implementing AMR in their applications, the benefits being better memory management and easier parallelisation due to the disjoint grids that are formed at each level, which makes it easier to split the domain among processors. The memory management in BSAMR is easier when compared to CBAMR as the hierarchy has to be maintained only for grids and not at a cell level. This can be complex in CBAMR if there are many cells that are refined. This motivates us in choosing the SAMR framework, specifically the BSAMR framework, for incorporating AMR into CRASH as well. Also, we have seen that the CHOMBO library is suitable for our goal due to its design, its scalability and the fact that it is already being used by hydro codes. With this aim in mind, we have developed CRASH-AMR, a novel implementation of CRASH which uses the CHOMBO library [2] to perform RT simulations on PBAMR grids generated by hydro codes. The topology of the grids is not changed within the RT simulation. This new feature of CRASH-AMR allows us to better resolve small scales in our radiative transfer simulations and to capture the ionisation and temperature patterns in high density structures.

In this Chapter we introduce CRASH, describing in detail the numerical scheme adopted to do MC RT simulations. We also take a look at the software architecture of the code, understanding the structure of the code is necessary to ensure that future developments are easily maintainable. We then discuss the technical details involved in the implementation of CRASH-AMR, keeping in mind the AMR framework in CHOMBO. We have tested the new release by performing the simulations established in the RTCCP [87], finding an excellent agreement between CRASH-AMR and the previous version of CRASH. We have also performed simulations using realistic density fields obtained from the Santa Barbara project showing that the new version of the code is able to accurately track the different size and shape of the ionised H II bubble in regions with increased resolutions. CRASH-AMR is hence able to provide a more accurate and detailed picture of the growth of H II regions through the different spatial scales involved in the reionisation process.

3.1 Radiative Transfer code CRASH

CRASH [40, 73, 122, 123, 124, 144, 146] is a 3D MC RT code that can self-consistently follow the formation and evolution in time of ionised regions created by sources present in a static and inhomogeneous gas environment consisting of H, He and metals; the gas temperature evolution is calculated self-consistently. The code can account for an arbitrary number of point sources as well as a UV background. CRASH has undergone a number of phases of code development; some of them were related to the addition of new physics modules for example [123, 124] and some were involved with improving the computational efficiency of the code through distributed memory parallelism [144]. The latest CRASH version, CRASH3, involved the addition of new physics as well as a major restructuring of the code [73]; this allows the addition of new physics or computational modules with ease. Our work is based on CRASH3 [see 73, and references therein for more details] and the developments presented here contribute to the new version CRASH-AMR by using CRASH3 as baseline but, for simplicity, without the inclusion of metals. In this section we discuss the MC scheme used in the code, and the RT effects that are accounted for during the simulation.

3.1.1 CRASH RT scheme

CRASH works by assigning the initial conditions (ICs) onto a static, regular 3D grid which specifies the gas number density n_{gas}, temperature T, the H, He ionisation fractions ($x_{\rm HII}, x_{\rm HeII}$ and $x_{\rm HeIII}$). The radiation field is accounted for using multiple point sources N_s, uniform UV background radiation or diffuse radiation from recombinations in the ionised gas. For the case with point sources, their locations are specified using cartesian coordinates, luminosity L and spectral energy distribution (SED) (S in erg $s^{-1}Hz^{-1}$). The background radiation is treated by assigning an intensity and SED over the entire domain. The radiation from each source is discretised into photon-packets represented by N_ν frequency bins, each containing $N_{p,\nu}$ photons as determined by the SED. The photon-packets that are emitted by the sources are propagated along the rays casted in random directions from the point sources. To emit the ray in a random direction, we make use of the standard *ran2* random number generator (RNG). A total simulation time t_s can be specified along with intermediate times for writing

out the relevant physical quantities. The simulation proceeds by emitting photon-packets from all the sources and propagating them along the rays until the end of simulation time.

We now look at the RT scheme implemented in CRASH in more detail.

Consider a point source with a time-dependent bolometric luminosity $L_s(t)$. For a total simulation time t_s, the total energy emitted by the source is

$$E_s = \int_0^{t_s} L_s(t)dt \tag{3.1}$$

This energy is distributed into N_p photon-packets, emitted by the point source at regular time interval $dt = t_s/N_p$. The time resolution of the simulation is hence decided by N_p. When multiple point sources are present, the time evolution of the ionising radiation field is reproduced by emitting a photon-packet from all the N_s sources at each time step $t_j = j\ dt$. Since a packet is emitted per time step, the number of time steps in the simulation is implicitly set to N_p. For each ray, associated with a photon-packet, we assign an angular direction described using spherical coordinates (r, θ, ϕ), and the origin of the ray is said to be at the emission cell (x_e, y_e, z_e). Given the origin coordinates and angular direction, the ray propagates in a direction given by

$$\begin{aligned} x &= x_e + r sin\theta cos\phi \\ y &= y_e + r sin\theta sin\phi \\ z &= z_e + r cos\theta \end{aligned} \tag{3.2}$$

We now look at ray propagation itself. Consider a ray along which a packet propagates by crossing a series of cells. For each cell l that is crossed, we calculate the casted path δ_l, i.e. the minimum distance the ray has to travel to exit the cell. To simulate the radiation-matter interaction, CRASH calculates the absorption probability of the photon-packet in the current cell as

$$P(\tau) = 1 - e^{-\tau} \tag{3.3}$$

where τ is the total optical depth of the cell due to the contribution of the different species, i.e.

$$\begin{aligned} \tau &= \tau_{\rm HI} + \tau_{\rm HeI} + \tau_{\rm HeII} \\ &= [\sigma_{\rm HI}(\nu) n_{\rm HI} + \sigma_{\rm HeI}(\nu) n_{\rm HeI} + \sigma_{\rm HeII}(\nu) n_{\rm HeII}]\delta_l, \end{aligned} \tag{3.4}$$

here n_A and σ_A are the number density and cross section of the absorber $A =$ H I, He I, He II respectively.

The number of photons absorbed in the cell l, N_A^l, is calculated as follows. A packet reaching the cell l has a photon content of

$$N_\gamma^l = N_\gamma^{l-1} - N_A^{l-1} = N_\gamma^{l-1} e^{-\tau^{l-1}} \leq N_{p,\nu} \tag{3.5}$$

If the packet reaches the cell with a photon content of N_γ^l, then the number of photons deposited in the cell is given by

$$N_\gamma^l = N_\gamma (1 - e^{-\tau}) \tag{3.6}$$

N_γ^l is then used to calculate the ionisation, recombination fractions and temperature equations that regulate the physical state of the gas (§2.4, [124]). The contribution of processes like photo-ionisation and photo-heating are treated using discretised equations involving N_γ^l, whereas recombinations, collisional ionisation and cooling are treated as continuous processes using respective rates (Appendix A, [124]). The code keeps a track of the integration time-step Δt, i.e. the time taken for two subsequent ray crossings through the same cell. This is used while calculating the change in ionisation fractions and other discretised quantities. For processes that are treated as continuous, the value of Δt should be much smaller than the characteristic time scales of these processes for all species

$$\Delta t \ll t_{min} = min[t_{recom,I}, t_{coll,A}, t_{cool}] \tag{3.7}$$

where $I = H^+, He^+, He^{++}$ and A is the absorber, as defined above. If this is not the case, the integration is split into n_s steps where $n_s = int[\frac{\Delta t}{f_s t_{min}}]$; here f_s is a fudge factor ranging from 50 - 100 to minimise any discretisation errors.

The angular direction of the ray and coordinates of the current cell are used to calculate the coordinates of the next cell that the ray will cross; this is repeated until the photon content in the packet is extinguished or, if periodic boundary conditions are not applied, the packet exits the grid. Note that once emitted, we do not change the direction of the photon-packet, i.e. it does not undergo any scattering.

CRASH has been used in cosmological set ups, where we need to account for the evolution of all physical quantities with redshift. This is done using a series of snapshots at successive redshifts, each with a different n_{gas} and point source configuration. The n_{gas} is evolved at different redshifts and the ICs for a snapshot are calculated self-consistently using the outputs from a previous snapshot. Since CRASH is used in a post-processing mode, the feedback effects of RT on the gas dynamics are not taken into account.

The numerical resolution within the code is determined by various factors, which we discuss in the next section.

3.1.1.1 Numerical resolution

We briefly mention here some of the factors that determine the numerical resolution of the code; these are important while setting up a CRASH simulation in order to get accurate results. These factors include

(i) N_s - number of point sources

(ii) N_p - number of photon packets emitted by each source and

(iii) N_c^3 - the number of grid cells

Let us consider a test case with N_s point sources, each emitting N_p photon-packets. Then the total number of photon packets emitted are $N_{tot} = N_p N_s$. If we have a grid of resolution N_c cells in each dimension, then the number of cells that each packet will cross is, approximately, $f_d N_c$ where f_d, a parameter, ranges in between $[N_c^{-1}, 1]$. f_d is ~ 1 for optically thin cases and $\sim N_c^{-1}$ for optically thick cases. Hence, the total number of times a cell will be crossed by the end of the simulation, N_{cr}, is given by

$$N_{cr} = N_{tot} \frac{f_d N_c}{N_c^3} = f_d \frac{N_s N_p}{N_c^2} \tag{3.8}$$

A user, while setting up the ICs of the simulation, must ensure that the value of N_{cr} is large enough such that the radiation field is appropriately sampled in each cell location. For black-body or power-law spectra, a value of $10^2 - 10^4$ crossings per cell was found to be sufficient enough for the radiation field to be well sampled [124]. Additionally the integration time-step Δt imposes a further restriction on minimum value of N_{cr}.

Finally, the computational costs associated with the code scale with N_s, N_p and N_c, these altogether determine the number of times the RT equations are solved during the simulation.

We next look at the architecture of the code to understand the software engineering techniques used to implement the RT scheme, in a structured and stable manner, in CRASH.

3.1.2 CRASH software architecture

We have discussed in detail the numerical method used in CRASH to implement RT. The aim of this thesis is to enable CRASH to carry out efficient RT simulations. However, before we move on to discussing how this can be done, we give an overall view of the software architecture of the CRASH code to get an idea of how additional modules can be added to the code. CRASH has undergone a number of phases of code development; our reference version CRASH3 has a different software architecture compared to the previous versions of CRASH. Henceforth, any future developments to the code will be done following the architectural framework implemented in CRASH3.

CRASH3 separates the main CRASH work flow from modules with different functionalities. It uses both control and data abstraction to manage the different modules involved. Each

module is isolated from the rest of the code, which not only simplifies code development, but also makes code debugging easier.

The workflow of the code is divided into the CRASH_SYSTEM and CRASH_SIMULATION software layers, the former being responsible for setting up the simulation using the different ICs. CRASH_SIMULATION on the other hand takes care of the main RT simulation. A number of logical switches, or controllers, allow the logical work-flow of the simulation to be changed. Hence the user can set up a problem using different input configuration files that are associated with different physical properties of the problem being looked at. This allows multiple RT simulations with different ICs to be run simultaneously.

To abstract the data, the code makes use of different Fortran datatypes that store information specific to a part of the RT simulation. For example, the datatype COSMOLOGICAL_BOX stores information specific to the physical cosmological box through which the photons are propagated. This includes the size of the box (in kpc or Mpc), the number of cells in each dimension and their size in *cm*. The physical variables related to the radiation field, for example, the point source spectrum and coordinates are stored in a datatype called SIM_RADIATION. The properties of the gas medium, for example, the temperature T and ionisation fractions of different species like H, He are stored in a datatype called SIM_GAS and the information pertaining to a photon-packet is stored in PHOTON_PACKET.

The outputs from an RT simulations can be written out at specific times, and the format of the output files can also be chosen. Some examples are HDF5, VTK and XML and binary formats. A number of post-processing tools are also available to analyse the simulation outputs.

We have now given an overview of the CRASH code, its numerical scheme and the software architecture of the code. Our aim is to be able to run RT simulations on AMR grids that have been generated by hydro codes. We next look at the techniques used to enable this in CRASH.

3.2 Enhancing CRASH RT simulations using CHOMBO AMR

In this Section, we describe how CRASH and CHOMBO have been interfaced to implement CRASH-AMR: we discuss both the adopted methodology and the solutions we found to the various technical issues that occurred during the code development. Throughout this Section, we will refer to the CHOMBO classes mentioned in Section 2.2.6.2 and describe how they are used in CRASH-AMR.

3.2.1 Interoperability between CRASH and CHOMBO

CRASH is originally implemented in Fortran 95 while CHOMBO is a C++ code. During the development of CRASH-AMR, we have moved to using features provided by the Fortran 2003 and some from the 2008 standard, to facilitate interfacing it with CHOMBO. We have created a C interface between Fortran and C++ to allow CHOMBO to communicate and share information with CRASH-AMR by using the interoperability features implemented in the programming

languages specifications. We use the grid representation of CHOMBO in CRASH-AMR to store the physical variables that have a spatial representation for e.g. $n_{\rm gas}, x_{\rm HII}, x_{\rm HeII}, x_{\rm HeIII}$ and T.

In the ray-tracing algorithm implemented in CRASH-AMR, the interaction of radiation with matter is computed in each crossed cell by solving the ionisation and temperature equations, this implies that the instances of the *Box* classes need to be continuously accessed during the propagation of the photon packets (see Section 3.1) to update and store the physical quantities through the multiple AMR levels provided by the hydro code. The computational cost of a continuous and inefficient access to the CHOMBO library could impact the global RT performances. Depending on the chosen resolution in space and the maximum refinement level provided by the gas dynamics simulation, a single ray could traverse in fact a large number of cells spanning different refinement levels, making the box iteration computationally inefficient when repeated for a large number of rays required by the MC convergence (typically $N_p \geq 10^7$). Note that this is not the way PBAMR based libraries are typically used in hydro-codes to access the information: the patch based scheme implemented in CHOMBO is in fact very efficient in the management of memory and parallel computational resources but provides information at the grid level instead of at the cell based quantities, as required by the CRASH-AMR RT scheme. A further complication arises from the fact that a realistic RT simulation generally involves an irregular distribution in space of the emitting sources from which a large number of rays is emitted in random directions, implying that the boxes at each refinement level are not accessed contiguously. As result, the standard interface provided by the CHOMBO library cannot be simply re-used in CRASH-AMR; to address this issue we have developed a new Fortran data structure, minimising the run-time overhead to access and iterate the AMR layers. We discuss this in the next Section.

3.2.2 Setting up CHOMBO based AMR hierarchy in CRASH

The new data structure that we have developed makes extensive use of some of the CHOMBO classes mentioned in Section 2.2.6.2, we mention these below.

- **In-memory data organisation and grid processing**. We use the *Box* class to represent CRASH data on a rectangular grid which can be coarsened or refined in specific regions. The disjoint array of boxes represented by the (*DisjointBoxLayout*) class provide access to each box at a certain refinement level. To iterate through these boxes, we use the *DataIterator* class.

  CRASH does RT using a ray-tracing algorithm, as mentioned in Section 3.1.1. To propagate the ray the algorithm requires the coordinates of the next cell that the ray passes through. An AMR grid can have multiple adjacent boxes at a certain refinement level. The coordinates of the next cell can lie in the same box or a neighboring box. So we need access to the neighbors of any particular box, which can be done using the *NeighborIterator* class.

  We also need to ensure that during ray-tracing, the ray is always at the highest refinement level possible for a particular location in the grid. For this, we need to know if a cell has been refined or not, and if so move the ray to a finer level. The *Box* class provides the *coarsen* and *refine* operations that coarsen or refine the box. We use this,

along with the *isEmpty* routine to determine if the intersection of a box with another box is empty. This indicates if the box lies within another box.

To store the data associated with the physical variables, in each grid cell, we used the Fortran Array Box class (*FArrayBox*).

- **AMR hierarchy management**. We have set up a class called `ChomboAMR` that inherits from the `AMR` class. This contains the complete AMR framework and manages each refinement level in the hierarchy, represented by the `AMRLevelCHOMBO` class. The latter inherits from the `AMRLevel` class in `CHOMBO` and contains information pertaining to the size of the domain at a particular level, the associated *DisjointBoxLayout* and operators from the *FineInterp* and `CoarseAverage` classes. `ChomboAMR` maintains a pointer to each refinement level for easy access to the relevant parent or child level.
- **Interactions between different refinement levels**. Since we use `CRASH-AMR` in the post-processing mode, operations of data smoothing are confined to the initialisation of the RT and are not performed during ray-tracing. To initialize the fine grid from the existing coarse grids we use the *FineInterp* class, while to update the coarse grids with the data on the finer grids the *CoarseAverage* class is adopted.
- **Data storage and grid I/O**. We use the HDF5 data format as done in `CHOMBO`.

During each ray traversal the photon-packet information has to be propagated through the AMR grid hierarchy from the highest refinement level to the coarse ones. At each step of the RT simulation, the data structure needs to know the refinement level a cell belongs to, whether the cell is refined or not, and which box contains the refined cell; the way all these quantities are accessed and the mapping between `CRASH` and `CHOMBO` data is described in the following paragraphs. Hereafter we will refer to the grid with the lowest resolution as "base grid", while the refined grids will be referred to as "refined levels".

Figure 3.1 shows the data representation in `CHOMBO` on the left-hand side (see also Section 2.2.6.2 for more details), and the `CRASH` equivalent on the right-hand. We also use similar colors in both sides to represent corresponding boxes, at a given refinement level. A simplified but representative `CHOMBO` hierarchy, consisting of the base level 0 and its refinements from 1 to $L-1$, is shown in the picture; for clarity purposes we just represent the boxes at levels 0, 1 and 2. To help the reader connecting this picture with the abstract data representation provided in Section 2, we note that each refinement level (dashed boxes on the left) is implemented in computer memory by an instance of the *AMRLevel* class, while the array of boxes at each level is implemented by the *DisjointBoxLayout* class. Each box, for example the B(0,0) at base level 0 (red box on the left), is an instance of the *Box* class.

The `CRASH` counterpart of the AMR hierarchy is mapped on the right-hand side of Figure 3.1: the base grid is represented as refinement level 0, while the L AMR grid levels are mapped with an array containing pointers to specific properties of each box. Also note that the boxes at each level are uniquely identified by an associated *localID* and *globalID*. As the number of boxes at each level is known to `CHOMBO`, we can have easy access to their information by just iterating though their *globalID*s. The corresponding *localID*, which is the index into the disjoint box array represented by the *DisjointBoxLayout* class, is used to get direct access to the data representing physical variables of the gas during the RT simulation. For example box

B(0,1) indicates that at refinement Level 1 there is a box with *localID* of 0 and a *globalID* of 1. The *localID* of 0 gives access to the box at the 0th index of the corresponding disjoint box array, and the relevant *FArrayBox* gives access to the physical data, via the Fortran pointers, pertaining to the box. Finally, the start and end coordinates of the box, which determine its size, are also stored to allow the ray-tracing algorithm to recognise if a ray has exited a box at a given refinement level [1].

The information stored above is done at the grid level. Ray-tracing happens on a "cell by cell" basis and we need some information at the cell level to ensure that the ray is always at the highest refinement level possible for a particular location in the grid. For this, we need to know if a cell has been refined or not, and if so move the ray to a finer level. We use the *coarsen*, *refine* and *isEmpty* operations to find if a box at level l, say $B(l,i)$ lies within a box at level $l-1$, say $B(l-1,i)$. If so, then for each cell in $B(l,i)$ we store the *globalID* of $B(l-1,i)$.

One important consequence of allowing multiple parents for a child grid is that it is not very efficient to create a tree structure for such a grid hierarchy. Given that we might not have a strict one-to-one relationship between a child and a parent grid, we have decided not to implement a tree structure to store the AMR hierarchy. Instead we use an array format that allows us to index into the data structure, at the right refinement level, to get the box properties.

We mentioned in Section 2.2.6.2 that the pointer returned by the *FArrayBox* class points to only one component in the grid. `CRASH-AMR`, in total needs ten components to store all the data needed for its RT calculations, for e.g. $n_{\rm gas}, x_{\rm HII}, x_{\rm HeII}, x_{\rm HeIII}$ and T and an additional two components to determine the child-parent relationship between the boxes at different refinement levels. So when we set up the data structure above, we set up an array of Fortran pointers that point to the data in each component of an *FArrayBox*. This is replicated for all boxes across different refinement levels.

The following paragraphs describe how the new data structure is used during the ray-tracing algorithm. First note that the sources emitting photons do not move across the grid during a RT simulation and then, in the notation established above, we only need to keep a track of the *globalID* of the box in which the source lies in. During the propagation of the photon packet, at each cell crossing, the following scenarios apply

(a) The ray might escape the grid and then we no longer follow it unless periodic boundary conditions are applied.

(b) The photon content of the packet is completely absorbed and then the propagation stops.

(c) The ray crosses the cell and enters a new cell at the same AMR level.

(d) The ray crosses the cell and enters another cell at a finer (or coarser) AMR level.

[1] Note that in a PBAMR scheme a refined box might lie over multiple coarse boxes, and so we need to keep a list of all parents. This is done by storing, for each box, the *globalIDs* of its parent(s), we also keep a list of its neighbor(s). Here the term neighbor(s) denotes all the boxes that are adjacent to a box at the same refinement level. Parent(s) refers to the box(es) at the coarser level that cover a particular box.

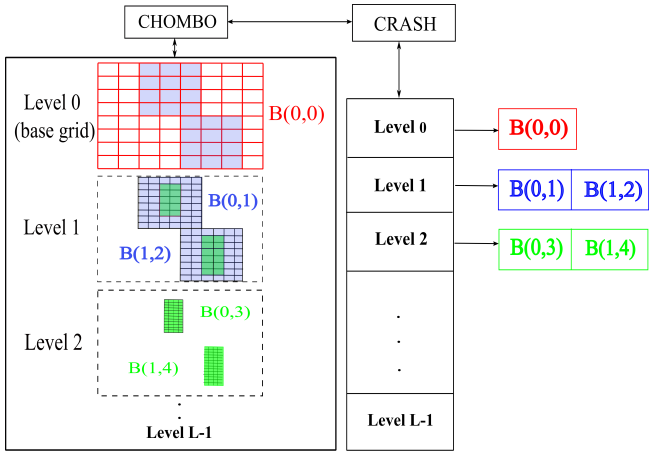

Figure 3.1 Interface between CRASH and CHOMBO, the grid hierarchy in CHOMBO being reflected through the data structure built in CRASH. The AMR grids, stored as an array of boxes in CHOMBO are stored in CRASH on a level basis, with pointers to the grid data at each level. Each box has an associated *localID* and *globalID*, see text for more details.

While case a and b do not need further comments, for case c, the new cell might lie in the same box or it might enter a new one. In the former case, we just continue the ray propagation as described earlier; in the latter case we need to use the new cell coordinates to inspect the neighbor list searching for the new box at the same refinement level. Case d finally needs a different approach because the cell optical depth depends on the refinement level the ray is crossing though the casted path (see Eq. 3.4). Here again different scenarios apply:

(1) The ray enters a finer level. In this case we use the *globalID* of box containing the child cell to move the ray to the finest level in which the cell is found. Note that the "proper nesting" criteria implies that while crossing levels, the ray can only move to the immediate child level, so we have to only search in the level above.

(2) The ray exits a box, as in c. In this case, we first use the neighbor list to search the new cell coordinates in the neighboring boxes. If not found we search in the parent list to find the new box. Once found we move the ray, as in case 1, to a finer level if the neighbor or parent is refined. Here as well, we only look in the levels immediately above or below to find the child or parent box.

The same procedure is repeated until cases a and b apply and the photon packet propagation stops. Once we get the *globalID* of the new box the ray is traversing, we use it to get the pointers to all the components within the corresponding *FArrayBox*. Figure 3.2 shows an example of rays propagating through multiple levels. The grid has three levels $L0$ (black box), $L1$ (blue boxes) and $L2$ (red boxes) with the point sources lying on $L2$. The rays being emitted from source $S1$ travel to a neighboring box at level $L2$ and to the parent level $L1$. The ray being emitted from source $S2$ travels from level $L2$ to $L1$ and back to level $L2$.

Some additional issues had to be kept in mind while developing CRASH-AMR, we discuss these in the following Section.

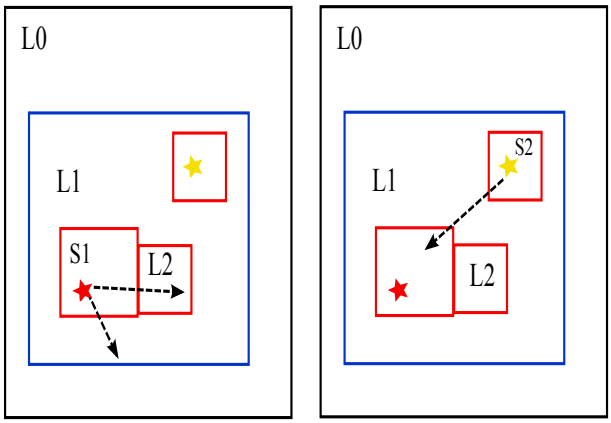

Figure 3.2 Illustration to show ray propagation across different refinement levels. The different levels $L0, L1$ and $L2$ are indicated by black, blue and red boxes respectively. Two sources $S1$ and $S2$ emit rays that travel through multiple levels.

3.2.3 Other technical considerations

It is imperative that we mention the other issues that need to be taken into account with the interface described in the previous sub-sections. The data structure built in CRASH uses a Fortran array at each refinement level to store the box properties. Fortran arrays can start with indexes other than 1, the minimum *globalID* at a particular level decides the start index of the array. For example, if the minimum *globalID* at level $L1$ is 10 and has 5 boxes, then the Fortran array is also allocated such that the first index is 10. This makes it easier to directly index into the array to get the necessary box.

We mentioned earlier that the data structure makes use of Fortran pointers to access the physical data stored in CHOMBO. At any given point of time in the RT simulation, the ray is passing through one unique box. Once we find the new box the ray is traversing through, we map its associated Fortran pointer to a 3D pointer in CRASH-AMR till the point the ray exits the box. The changes made to the physical data during this period are hence reflected on the CHOMBO side as well. However we need to consider certain technical details while mapping these pointers.

Fortran pointers, by default, start with an index of 1, unless declared otherwise. These pointers are mapped to a component in the *FArrayBox*, which can have different start coordinates depending on its position in the 3D grid. As a result we need to make sure that we refer to the right cell coordinates while propagating the ray. We keep a track of the 3D grid coordinates at each cell crossing, this is used to check if the ray has exited a box or grid. The same grid coordinates are then used to find the offset into a particular *FArrayBox*, i.e. the Fortran pointer currently mapped to the data in the grid that the ray is traversing. This gives us access to the physical data in the cell and also determines if we need to search for a new box or stop ray traversal.

It is important to note that although we use CHOMBO to initialise and store the AMR grid, once this data has been mapped on to the CRASH side, our implementation does not call any CHOMBO routines during the RT simulation. The data structure is used purely to cross the

levels and have a fast access to the grid data. As a consequence of this architectural choice, there is no overhead of using CHOMBO during the ray-tracing routine but the time needed to find the right cell. We do call the library to write out the output files at pre-determined times during the simulation. However the total run time, when compared to CRASH3, increases by only by 3% due to the use of CHOMBO. This can be attributed to the additional time taken to set up the AMR grids in CHOMBO and the Fortran data structure in CRASH-AMR. Since this is done once at the start of the simulation it is not a major impediment to the code.

The I/O in CHOMBO is done by making calls to the HDF5 library and passing each box at a refinement level as data to be written out. In addition to this, some metadata pertaining to the domain size, its periodicity and the names of all components are also written out. We have made changes to these routines so that the component names are not generic, for example, "component_0" but reflect the physical variable being referred to, for example, "H Ionisation fraction". The structure of the output files, with respect to the metadata, is now similar to that obtained from the hydro code. The read routines in CHOMBO have also been changed to reflect the component names being read in.

This completes an overview of CRASH-AMR which has been developed following the same software architecture as CRASH3. We look at some relevant details regarding the architecture of the code.

3.2.4 Software architecture of CRASH-AMR

In Section 3.1.2 we had discussed the software architecture of CRASH3. The development of CRASH-AMR has also been done in similar lines. Similar to CRASH3 we use control abstraction to change the work flow of the code depending on whether the controller for the AMR module is enabled or not. The user can compile the code with or without the AMR functionality and enable or disable it during run time. When enabled, the code uses the AMR grids to do RT simulations, the traditional method of data storage and setting up of simulation ICs only on the base grid is done otherwise. If the AMR functionality is enabled then the user can either run simple tests with pre-defined refinement criteria or more realistic test cases with AMR grids provided by hydro codes, as shown in Section 3.3. By adopting pre-defined refinement criteria, the user could decide, for example, to refine an arbitrary part of the base grid and set up specific test cases, while for realistic gas configurations, the refinement is generally determined by the hydro code and CRASH-AMR operates in a post-processing mode.

To generate the ICs mentioned we have developed a number of pre-processing tools; in addition to these we also provide other options

- Options to generate point sources, at a certain location and refinement level, with a given reference luminosity using high density peaks
- Gather point sources located at higher refinement levels on to the levels below
- Extract the necessary data from a hydro simulation output to generate input files for CRASH-AMR
- Create uniform resolution grids from AMR grids, to compare the results of the RT simulation with and without using AMR grids

These have been used for the tests mentioned in Section 3.3. In addition to the above, we have also developed some post-processing tools for analysing the simulation outputs:

- Option to calculate the spherical average of a physical quantity like $x_{\rm HII}$ and T for comparing the results with `CRASH3`
- Combine two HDF5 files so that contour plots can be created
- Creating Line of Sights (LOSes) away from the point source to understand the shape and extent of the H II region around the point source

Similar to the data abstraction methodology followed in `CRASH3`, we define a new datatype `AMR_PARAM` that contains general information about the AMR hierarchy, for example the number of refinement levels, the domain size, refinement ratios between levels and periodicity information. To reduce the number of accesses to the data structure shown in Figure 3.1 we define another datatype `AMR_BOX` that contains the box properties of the box the ray is currently traversing through. We use this to check if the ray has exited a box and when a new box is found, its copied to `AMR_BOX`.

This completes our description of the new code `CRASH-AMR`, we now look at the various tests scenarios set up to test the code.

3.3 Test scenarios and Results

In this section we show the results of some of the tests we have performed to guarantee the reliability of the new AMR implementation. We run a number of test cases in idealised configurations, in Test 1 we compare `CRASH-AMR` with the AMR functionality disabled to `CRASH3`, then we compare results with/without the AMR functionality enabled. These set-ups are useful to check the numerical noise introduced by the presence of the AMR grids on the RT algorithm. In Test 2, we apply `CRASH-AMR` to a realistic density field from the `CHARM` simulations described in [128]. Henceforth, we use d to represent the comoving distance from a point source, the units in kpc or Mpc are indicated accordingly.

3.3.1 Test 1: Strömgren sphere in a H medium

We have set up a test equivalent to Test 1 of the RTCCP. The test simulates the evolution of an ionised region around a single point source located at the grid origin (0,0,0) in a cosmological box of side length $L_{box} = 6.6$ kpc (comoving) and mapped on a grid of 128^3 cells. The source is assumed to be steady with an ionising rate of $\dot{N}_\gamma = 5 \cdot 10^{48}$ photons s^{-1} and an associated monochromatic spectrum with $h\nu = 13.6$eV. The volume is filled by a uniform and static gas of number density, $n_{\rm gas} = 10^{-3}$ cm^{-3}, containing only H. The gas is assumed to have an initial ionization fraction (given by collisional equilibrium) $x_{\rm HII} = 1.2 \cdot 10^{-3}$, the gas temperature T is fixed at 10^4 K with a simulation time $t_{sim} = 500$ Myr, starting at redshift $z = 0.1$. We output the results at intermediate times t=10, 50, 100, 200 and 500 Myr as in the original set-up. A well defined analytical solution for calculating the position of the ionisation front measured at fixed $x_{HII} = 0.5$ (I-front), at different times in the simulation, exists for this set-up (§3.2, [87]).

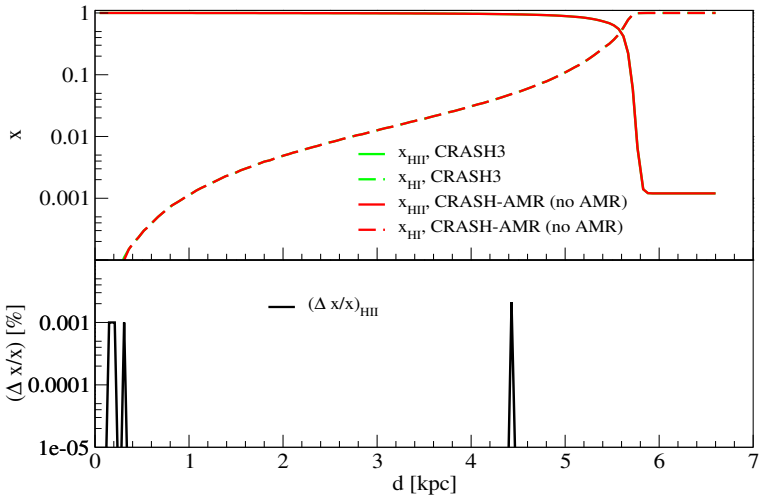

Figure 3.3 Spherically-averaged profile at time $t = 500$ Myr, at d [kpc], for Test 1a. The colors refer to CRASH3 (green) and CRASH-AMR (no AMR) (red). **Top:** Profiles of $x_{\rm HII}$ (solid lines) and $x_{\rm HI}$ (dashed lines). **Bottom:** Δ between the CRASH3 and CRASH-AMR (no AMR) results.

3.3.1.1 Test 1a: AMR disabled

To verify that the changes done to enable RT on AMR grids do not introduce any numerical noise, we have run Test 1 with AMR disabled in CRASH-AMR and compared the results to those from CRASH3.

The outcome is shown in Figure 3.3, where the panel shows the spherically-averaged physical quantities as a function of d [kpc], together with the percentage difference (Δ) between the CRASH3 and the CRASH-AMR (no AMR) results. We define $\Delta = (R_{ref} - R_i) \cdot 100 \ / \ R_{ref}$ where R_{ref} refers to results of CRASH3 and R_i refers to results of CRASH-AMR. We find an excellent agreement between the results, the size of the ionised region in both cases extends up to 5.4 kpc and drops thereafter, this agrees with the analytical solution given in [87] for the Strömgren radius. The spikes that we see in Δ are due to numerical artefacts caused by optimisation of CRASH-AMR involving rearranging of floating point arithmetic expressions and are not due to the changes associated with CHOMBO. This shows that the AMR feature in CRASH-AMR is isolated from the rest of the code and can be disabled without introducing any numerical noise into the results.

3.3.1.2 Test 1b: AMR enabled

Our next step has been to run the previous test, with the AMR functionality enabled. We tag and refine, by a refinement factor of 2, 100^3 cells around the source placed at the origin. The ICs are set up on the base grid as well as on the refined grid. The source is located at the highest refinement level which represents a domain of 256^3. Our choice of refining a region of 100^3 cells ensures that it is large enough ($\sim$5.15 kpc each side) to represent the $x_{\rm HII}$ region at a higher resolution, but also allows to check that the ray-tracing code works correctly.

With such a set-up there will be rays that traverse from the finer to coarser levels and the ray-tracing routine should take into account the change in grid levels while calculating the optical depth. We compare the results of this test with those from CRASH-AMR in § 3.3.1.1. Note that we compare results between a set up where the finest grid level represents a domain of 256^3 and the other represents a domain of 128^3. This is intentional since we want to show that for a test case which has an analytical solution, CRASH-AMR can be run with a higher resolution in the region of interest and provide comparable results with a set up that uses only one grid level.

Figure 3.4 shows the spherically-averaged physical quantities as a function of d [kpc], together with the Δ values calculated using R_{ref} for results of CRASH-AMR (no AMR), R_i refers to the results of CRASH-AMR with one refinement level (1 r.l.) results. The $x_{\rm HII}$ region for CRASH-AMR extends till 5.4 kpc, as per the analytical solution given in [87]. We find that Δ is very high, $\sim 1000\%$, at a distance of 5.9 kpc and drops thereafter. The differences that we see are not due to the implementation in CRASH-AMR, but rather they are associated to the higher grid resolution in the refined region. When we have a grid at a higher refinement level all the refined cells that cover a coarse cell might not lie within a given radius from the source and thus not contribute to the spherical average.

We mentioned earlier that the setups described in Test 1a and 1b have a well defined analytical solution for calculating r_I, which is given by

$$r_I = r_S[1 - \exp(-t/t_{rec})]^{1/3} \tag{3.9}$$

where r_S is the Strömgren radius, which in our case is 5.4 kpc and t_{rec} is the recombination time for H given by

$$t_{rec} = [\alpha(T) x_{\rm HII}]^{-1} \tag{3.10}$$

Assuming $\alpha(T)$, the recombination rate for H, is $2.59 \cdot 10^{-13} {\rm cm}^{-3}{\rm s}^{-1}$, t_{rec} is 122.4 Myr. Figure 3.5 shows r_I at times t=10, 50, 100, 200 and 500 Myr between the analytical solution and CRASH3, CRASH-AMR results. In case of CRASH-AMR we show the results with AMR disabled and with one r.l. The bottom panel shows the Δ values calculated using R_{ref} for the analytical solution, R_i refers to the results of CRASH3 and CRASH-AMR. We find that, by the end of the simulation where the codes reach convergence, the difference in the I-front position between the analytical solution and the codes is only 4%. Between CRASH3 and CRASH-AMR, we do not find any difference in the results.

Our next test is similar to Test 1 but takes into account the presence of He as well.

3.3.2 Test 2: Strömgren sphere in a H+He medium

We have set up a test equivalent to Test 2 of RTCCP [87]. The test simulates the evolution of an ionised region around a single point source located at the grid origin (0,0,0) in a cosmological box of side length $L_{box} = 6.6$ kpc (comoving) and mapped on a grid of 128^3 cells.

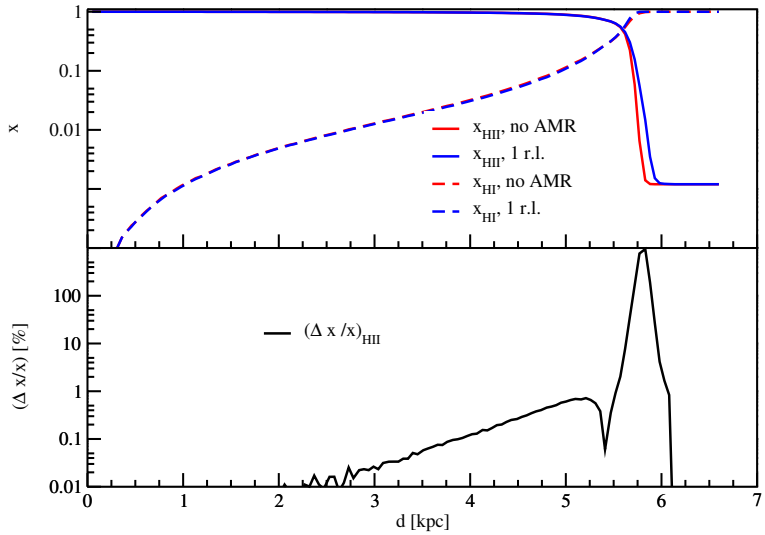

Figure 3.4 Spherically-averaged profile at time $t = 500$ Myr, at d [kpc], for Test 1b. The colors refer to CRASH-AMR (no AMR) (red) and CRASH-AMR with one refinement level (1 r.l.) (blue). **Top:** Profiles of $x_{\rm HII}$ (solid lines) and $x_{\rm HI}$ (dashed lines). **Bottom:** Δ between the CRASH-AMR (no AMR) and CRASH-AMR (1 r.l.), results.

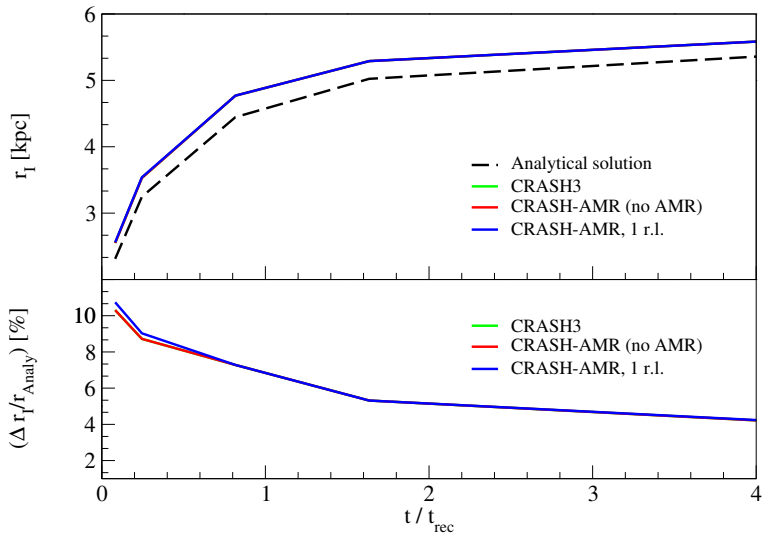

Figure 3.5 Radius of the I-front, r_I, at times t/t_{rec}, where $t = 10,\ 50,\ 100,\ 200$ and 500 Myr and $t_{rec} = 122.4$ Myr, for Test 1a and 1b. The colors refer to the the analytical solution (black), CRASH3 (green), CRASH-AMR (no AMR) (red) and CRASH-AMR with one refinement level (1 r.l.) (blue). **Top:** r_I for the analytical solution (dashed line) and CRASH3, CRASH-AMR (solid lines). **Bottom:** Δ between the analytical solution and CRASH3, CRASH-AMR (no AMR) and CRASH-AMR (1 r.l.) results.

The source is assumed to be steady with an ionising rate of $\dot{N}_\gamma = 5 \cdot 10^{48}$ photons s^{-1} and an associated black-body spectrum at temperature $T_{BB} = 10^5$ K. The volume is filled by a uniform and static gas of number density, $n_{\rm gas} = 10^{-3}$ cm^{-3}, containing H (92%, by number) and He (8%). The gas is assumed to be fully neutral, the gas temperature T is initially set to 100 K and it is calculated self-consistently with the progress of ionisation for a simulation time t_{sim} = 500 Myr, starting at redshift $z = 0.1$. We output the results at intermediate times t=10, 50, 100, 200 and 500 Myr as in the original set-up.

3.3.2.1 Test 2a: AMR disabled

Similar to Section 3.3.1.1, we have run Test 2 to verify that the changes done for `CRASH-AMR` do not introduce any numerical noise. We have run Test 2 with AMR disabled in `CRASH-AMR` and compared the results to those from `CRASH3`.

The outcome is given in Figure 3.6, where each panel shows spherically-averaged physical quantities as a function of d [kpc], together with Δ values, calculated using R_{ref} for results of `CRASH3` and R_i for results of `CRASH-AMR`. From the Figure it is clear that there is no difference between the two codes. As mentioned in 3.3.1.1 the spikes that we see in Δ are due to numerical artefacts caused by code optimisation.

3.3.2.2 Test 2b: AMR enabled

Our next step has been to run the previous test with AMR enabled. We set the ICs on a grid similar to that in Test 1b. The results are shown in Figure 3.7, where each panel shows the spherically-averaged physical quantities as a function of d, together with the Δ values between the `CRASH-AMR` (no AMR) and `CRASH-AMR` (1 r.l.) results. Δ in this case uses R_{ref} as results of `CRASH-AMR` (no AMR), while R_i refers to the results of `CRASH-AMR` (1 r.l.). Here we find that Δ can be as high as 10 % for $x_{\rm HI}$ and $x_{\rm HeII}$ in a handful of cells near the source ($\sim$ 0 - 1 kpc), but more typical values do not exceed 1 % for all the physical quantities.

As mentioned in Section 3.3.1.1 the differences seen are not due to the implementation in `CRASH-AMR`, but are due to the way the spherical average is calculated. With a grid at a higher refinement level all the refined cells that cover a coarse cell might not lie within a given radius from the source and thus not contribute to the spherical average.

The code shows excellent results for the standard test cases prescribed in [87]. We provide some additional tests showing the dependence of the results on the grid resolution in Section 3.4. Next, we look at some tests done with a realistic density field obtained from a hydro simulation.

3.3.3 Test 3: a realistic density field

In this section we apply `CRASH-AMR` on a density field snapshot obtained from a simulation defined by the "Santa Barbara Cluster Comparison Project", where the formation of a galaxy cluster in a standard CDM universe is simulated [61]. The simulation has been performed by

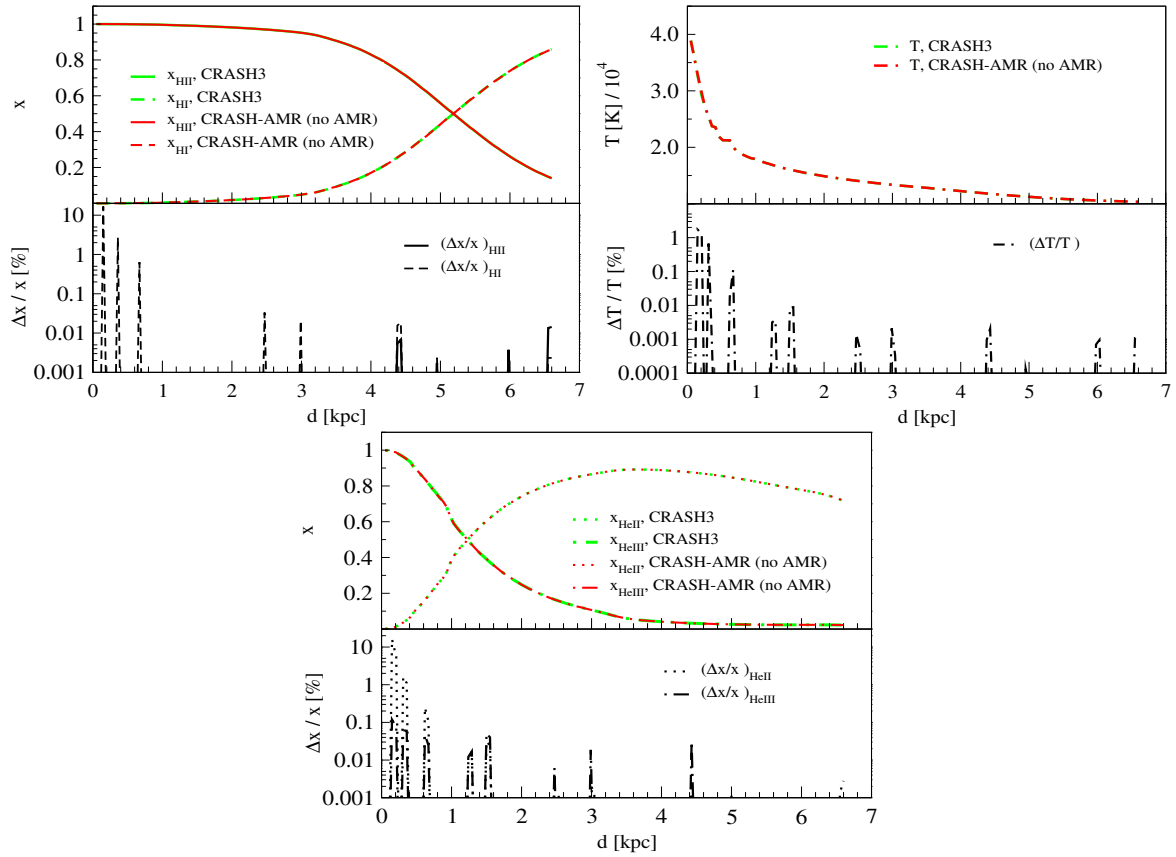

Figure 3.6 Spherically-averaged profiles at time $t = 500$ Myr, at d [kpc], for Test 2a. The colors refer to CRASH3 (green) and CRASH-AMR (no AMR) (red). **Top left:** Profiles of $x_{\rm HII}$ (solid lines) and $x_{\rm HI}$ (dashed lines). **Top right:** Profile of T (dash-dot lines). **Bottom:** Profiles of $x_{\rm HeII}$ (dotted lines) and $x_{\rm HeIII}$ (dot-dash lines). The bottom sub-panels show Δ between the CRASH3 and CRASH-AMR (no AMR) results.

the hydro code CHARM [128] on a simulation box size $L_{box} = 64$ Mpc (comoving) at redshift $z = 0.1$. The cosmological parameters were $\Omega_m = 1, \Omega_b = 0.1$, $\Omega_l = 0$ and $H_0 = 50 \mathrm{kms}^{-1}\mathrm{Mpc}^{-1}$. The simulation is initialised at $z = 40$ with two grids, a base grid of 64^3 cells representing a box of 64^3 Mpc, and a grid of 128^3 cells placed at the centre of the base grid representing a box of 32 Mpc on each side. Only the central region is refined based on a local density criterion, with a refinement ratio $n_{ref} = 2$. At the end of the simulation there are six refinement levels in total along with the base grid. The cell width at the coarsest level is 1 Mpc while that at the finest level, with a resolution of 4096^3 cells, is 15 kpc (comoving). The code CHARM adopts the CHOMBO library to implement the AMR functionality, and the HDF5 files available from the output of this simulation can be immediately used as an input to CRASH-AMR by extracting the necessary information from the HDF5 metadata. As the simulation does not provide information on the star formation, we define the point source locations associating them with the gas density peaks at the most refined level.

We set up the following simulations

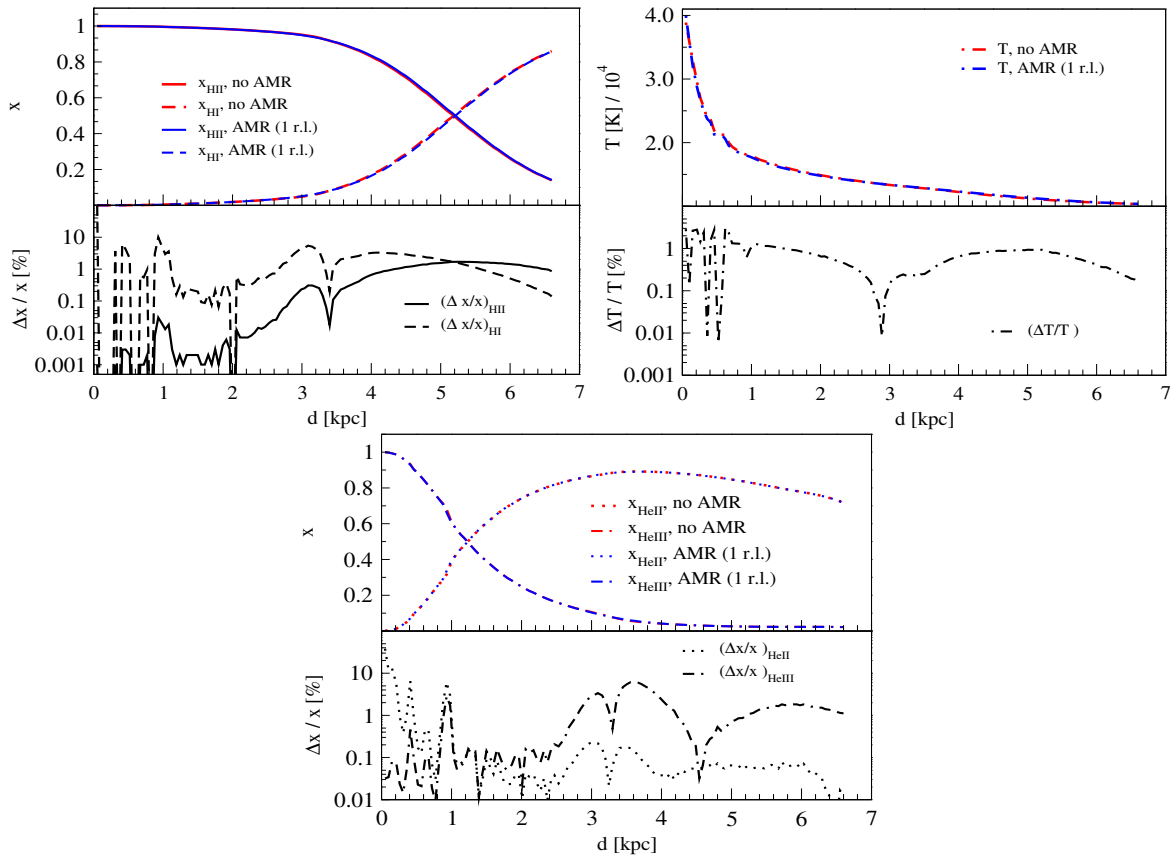

Figure 3.7 Spherically-averaged profiles at time $t = 500$ Myr, at d [kpc], for Test 2b. The colors refer to `CRASH-AMR` (no AMR) (red) and `CRASH-AMR` with one refinement level (1 r.l.) (blue). **Top left:** Profiles of $x_{\rm HII}$ (solid lines) and $x_{\rm HI}$ (dashed lines). **Top right:** Profile of T (dash-dot lines). **Bottom:** Profiles of $x_{\rm HeII}$ (dotted lines) and $x_{\rm HeIII}$ (dot-dash lines). The bottom sub-panels show Δ between the `CRASH-AMR` (no AMR) and `CRASH-AMR` (1 r.l.), results.

(a) multiple point sources at locations far enough so that they do not gather at the lower refinement levels, monochromatic sources, T kept constant;

(b) multiple point sources at locations close enough so that they can be gathered at the lower refinement levels.

We set a reference ionisation rate, $\dot{N}_{\gamma,ref}$, for the source at the highest gas density peak. For the other sources i, $\dot{N}_{\gamma,i}$ was set as:

$$\dot{N}_{\gamma} = \frac{\dot{N}_{\gamma,ref} \cdot m_i}{m_{ref}}, \tag{3.11}$$

where m_{ref} and m_i are the mass in the cell containing the reference source and source i, respectively. The initial temperature is $T = 100$ K and gas is assumed to be fully neutral.

To emphasise the advantage of an AMR scheme, we compare results of simulations run with different refinement levels. Additionally, as mentioned above, the sources are located at

the highest refinement level, so that if one or more of them lie within the same cell at the coarser levels, we consider them to be a single source with luminosity given by the sum of the corresponding luminosities at the finest level.

To ensure a good convergence of the MC code, we sample the radiation field with a number of photon packets high enough to reach convergence for each test case run at different refinement levels. We find that the MC scheme converges with 10^8 photon packets per source (0.07% difference in volume averaged $x_{\rm HII}$ values between two test cases with 10^8 and 10^9 photon packets per source). However the convergence of the MC scheme is very much problem dependent and depends on multiple factors, hence we do not discuss this in more detail for the test cases we present in the following sections.

3.3.3.1 Test 3a: multiple sources set far apart

Here we place twenty point sources far from each other enough to remain single sources at all refinement levels. This configuration would test the effect of grid resolution on the RT simulation. We adopt $\dot{N}_{\gamma,ref} = 8 \cdot 10^{53}$ photons s^{-1} (3.11), and the total simulation time is 500 Myrs. For simplicity, here we consider a H only gas configuration and each point source is monochromatic with $E_\nu = 13.6$ eV. The T is initially set to 100 K and is kept constant throughout the simulation.

Figure 3.8 shows the maps of $x_{\rm HII}$ created in simulations with different refinement levels (from 3rd to 6th level, from left to right), at simulation times $t = 100,\ 200$ and 500 Myr (from top panel). In the bottom panels we also show the gas number density fields ($n_{\rm gas}$), as gradient from black (low density) to white. Dotted lines represent the extent of the different refinement levels (see the caption for more details). For reference, we also show the location of the most luminous point source. Note that the RT is done in post-processing and the gas configuration does not change during the ionisation evolution.

At $t = 100$ Myr (top panels), the $x_{\rm HII}$ maps do not show much differences, being dominated by the most luminous source which easily maintain the ionisation against the progressively steeper density gradients introduced by AMR. Sensitive differences start to get more visible at $t = 200$ Myr, where separate bubbles can be seen on the right side of the box as we go to higher refinement levels. The largest differences are finally seen at the final time $t = 500$ Myr. First note that we cannot observe a direct correlation between position of refinement levels and distortions in the ionisation pattern (compare ionisation pictures with gas number density) because the sources in the plane are able to easily maintain their surrounding regions fully ionised against the various density gradients. Second, note that the extent of the fully and partially ionised H II regions at different resolutions shows obvious differences: they get smaller and sharper with higher resolution and do not tail away, as shown instead at lower resolutions. For example, the ionisation front, measured at fixed $x_{HII} = 0.5$, extends up to the left end of the box for the case with 3 refinements, but is much more smaller for the case with 6 refinements. This is due to the larger density gradients present at higher resolutions, which makes the escape of ionising photons difficult, and delays the ionisation fronts.

Finally note that the presence of multiple point sources on different planes of the cube, and resolved by different AMR layers, creates an intricate combination of three-dimensional RT

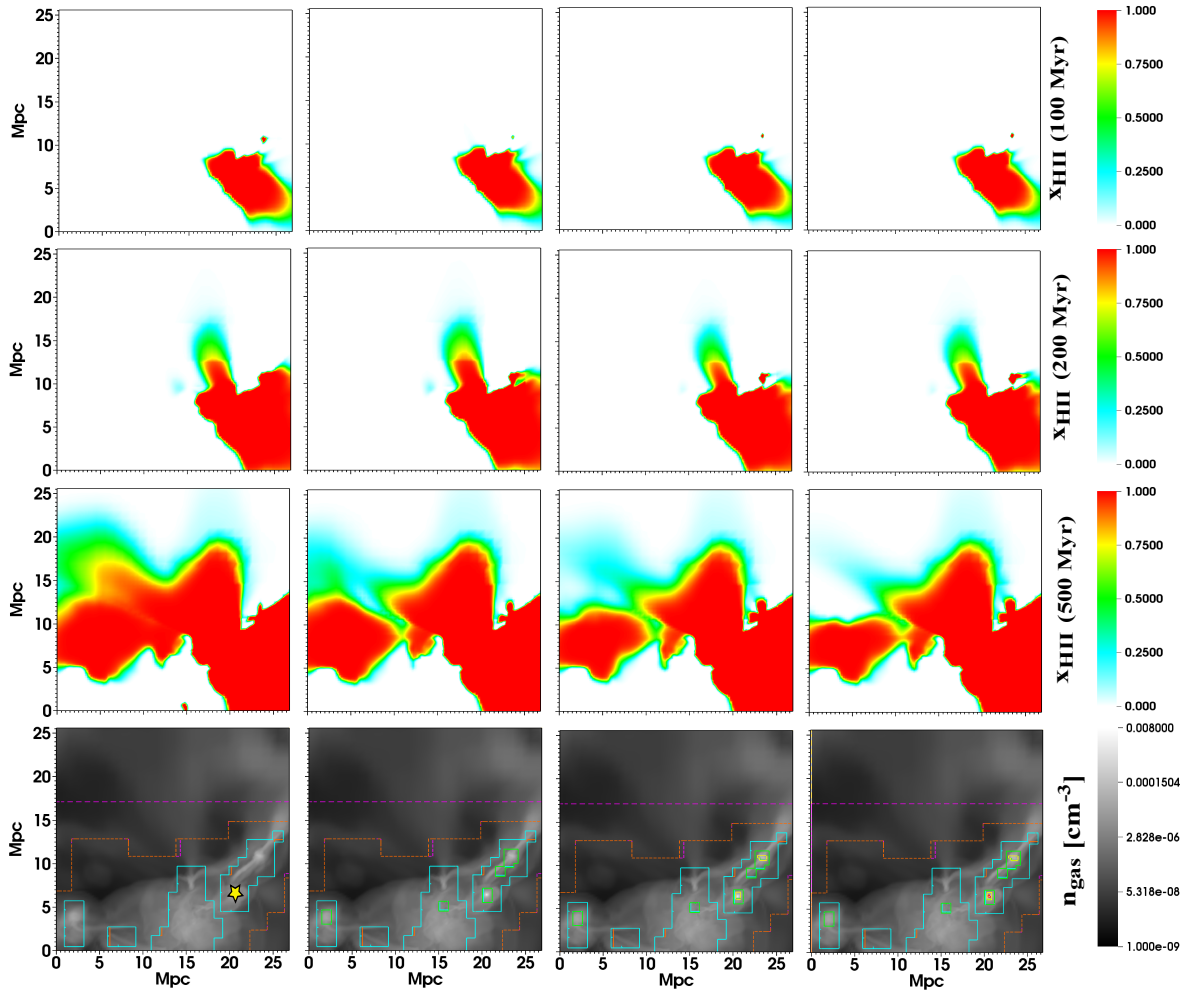

Figure 3.8 Maps cut through the simulation volume for Test 3a. **Top:** Maps of $x_{\rm HII}$ at time $t = 100$ Myr. **Second from top:** Maps of $x_{\rm HII}$ at time $t = 200$ Myr. **Third from top:** Maps of $x_{\rm HII}$ at time $t = 500$ Myr. **Bottom:** Maps of $n_{\rm gas}$, the dotted lines represent the extent of the different refinement levels associated with $n_{\rm gas}$. The colours refer to different refinement levels, light orange (base grid - not seen here), pink (1st r.l.), orange (2nd r.l.), blue (3rd r.l.), green (4th r.l.), yellow (5th r.l.) and red (6th r.l.). From left to right, the columns refer to simulations run with three, four, five and six refinement levels (see text for more details). The width of each slice is 25 Mpc.

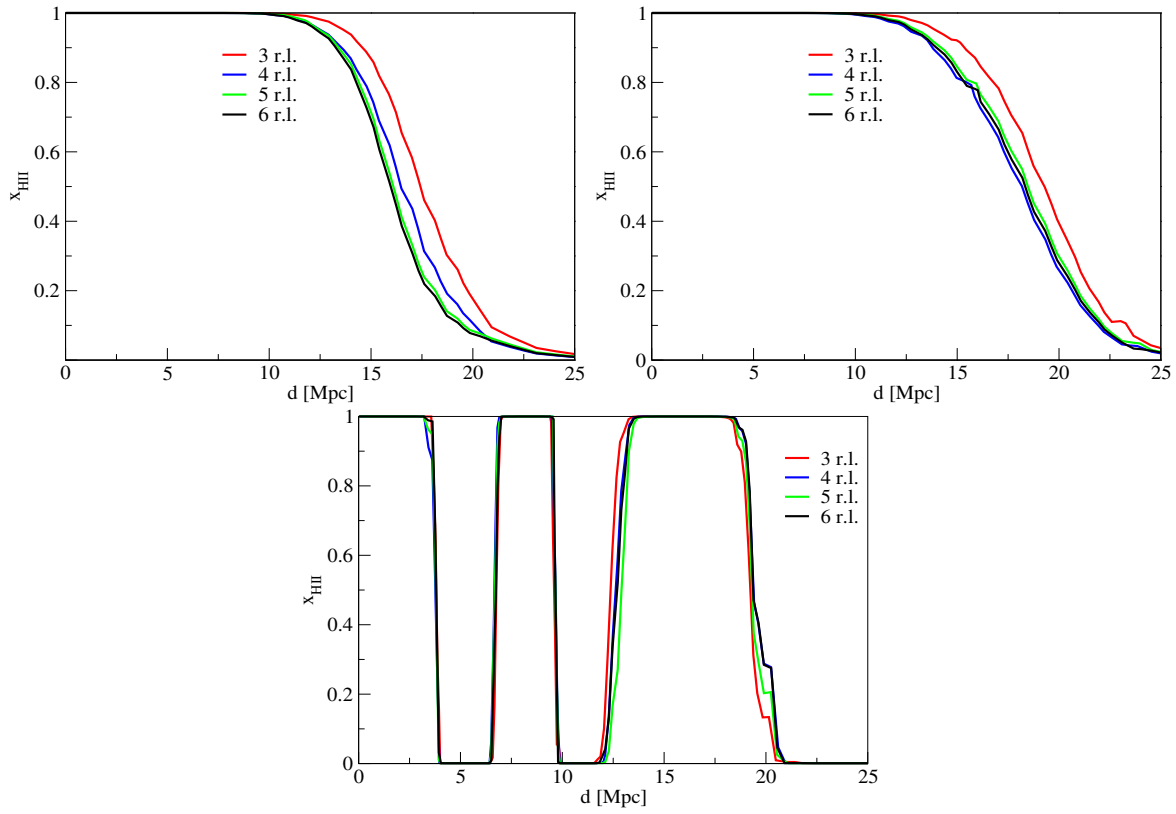

Figure 3.9 LOSes in random directions when moving away from a point source, for Test 3a at time $t = 500$ Myr. The colors refer to a test case with three (red), four (blue), five (green) and six (black) r.l.

effects in the final configuration of the overlapping of H II fronts. This is more evident in the case with 5 and 6 refinements, in two distinct regions. The H II region on the left is in fact formed by a point source lying on a different plane in the simulation box, and evolves differently with the increase of the AMR levels. This provides a final bubble distribution and overlap in space, which is very sensitive to the number of adopted AMR refinements. These different ionisation pictures also translate into quantifiable volume averaged properties of the ionisation pattern: a H II fraction of 4.94, 4.74, 4.51 and 4.39 $\cdot 10^{-2}$ is estimated for the 3rd, 4th, 5th and 6th refinement levels, respectively, with a 12.5% difference between the 3rd and 6th levels.

Figure 3.9 shows three line of sights (LOSs) along random directions when moving away from the most luminous point source in the simulation; for each LOS we plot the $x_{\rm HII}$ values.

The trend found in the three LOSs is clear and reflects the global considerations already made when commenting on Figure 3.8. In the first two panels the I-front systematically recedes when a deepest layer is accounted for by an additional refinement level (see different colour lines in the panel from red to black) due to the increased gradient in $n_{\rm gas}$ and the H recombination rate which is higher by a factor of 3.5, between the cases with 6 and 3 refinements, in the fixed T setup of this run. Also, the width of the I-front between the two cases is smaller by 9 and 5% respectively, for the two panels. We finally note that the extent

of the fully ionised region (measured as distance d at which $x_{\rm HI}$ drops below 0.1) is almost the same in all cases, with a maximum difference of 10% between the cases with 3 and 6 refinement levels.

The third panel shows three ionised regions along the selected LOS: the first two are fully ionised and do not show a sensitive difference between the refinement levels, while the most distant shows a trend similar to the other two panels. The extent of the third ionised region is in fact larger for the lowest resolution case and progressively smaller at increasing resolution, while its variation is less evident than in the two previous panels. In conclusion, by analysing three different LOSs, we find a trend similar to what was discussed for Figure 3.8: high density regions near the bright sources are easily ionised despite additional AMR refinements, while at larger d, other high density regions remain self shielded from the global radiation field and are neutral. Low density regions, also far from the source, are instead ionised, probably from nearby point sources or some global background established locally, at final simulation time.

We then conclude that the photo-ionisation algorithm of `CRASH-AMR` is highly sensitive to the variations of gas number density gradients progressively resolved by an increasing number of refinement levels; this is immediately reflected in the variations in patterns of the H ionisation fronts. `CRASH-AMR` will provide a more precise and realistic representation of how the ionised bubbles form and grow around the high density regions in which star formation occurs in galaxies, as well as a better estimate of the escape of ionising photons through the IGM when local-scale reionisation simulations, will be performed with this technique.

3.3.3.2 Test 3b: multiple sources set close to each other

Differently from Test 3a, here we place the twenty sources so that they are close enough at the highest resolution to be gathered at lower refinement levels. This results in 12, 6 and 2 sources at the 5th, 4th and 3rd refinement level, respectively. We assume $\dot{N}_{\gamma_{ref}} = 5 \cdot 10^{53}$ photons s^{-1} (3.11). The rest of the ICs are the same as Test 3a.

The set-up of this test is such that multiple sources are represented at the coarser levels as a single one of higher luminosity. As a result, we expect to see the growth of only one ionised region at the coarser level whereas at higher levels the ionised regions are distinct from each other, at least at early simulation times, and they would join only later on, once the regions have grown in size.

Figure 3.10 shows maps of $x_{\rm HII}$ for the different refinement levels in the simulations at time $t = 10$ Myr. We find a single ionised region at low resolution but multiple, disjoint regions at higher resolutions. As in previous cases, the ionised region is smaller at higher resolutions. This translates into a volume averaged H II fraction of 1.38, 1.43, 1.38 and 1.37 $\cdot 10^{-3}$ for the 3rd, 4th, 5th and 6th refinement level, respectively, with a .7% difference between the 3rd and 6th levels. At 500 Myr, the differences in the H II fraction averages are 1.1%.

Figure 3.11 shows the same LOSs in random directions when moving away from a point source, for $x_{\rm HII}$ at 500 Myr, for the different test cases. We find a trend similar to Test 3a, with the largest differences observed in the partially ionised gas. Since the T is fixed for this setup as well, and the density maps are the same as in Test 3a, the recombination rate

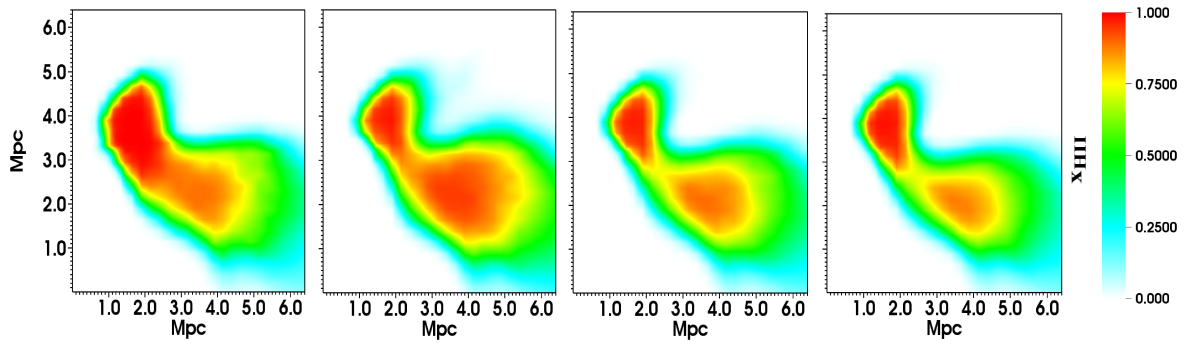

Figure 3.10 Maps of $x_{\rm HII}$ for Test 2c at time $t = 10$ Myr. From left to right, the columns refer to simulations run with three, four, five and six r.l. (see text for more details). The width of each slice is 6 Mpc.

increases by a factor of 3.5 at the highest refinement when compared to that for the lowest refinement. Also the width of the I-front, for the case with 6 refinements, is smaller by 2, 28 and 3% respectively in the three panels, when compared to the case with 3 refinements. Here as well, the differences we see in the ionised regions are due to the increasing grid resolution and better representation of over density in the grid due to the adaptivity in the hydro code.

We have tested and verified the new code CRASH-AMR for standard test cases and with a realistic test, obtaining excellent results in the process. Comparing the results with CRASH3 shows some differences, which are due to the code optimisations done in the new version. We look at some additional test cases to ensure that this is the case.

3.4 Dependence on grid resolution

The difference in results that we see in Sections 3.3.1.2 and 3.3.2.2 are due to the grid resolution only and not the implementation in CRASH-AMR. To demonstrate this, we have set up test cases, similar to Test 2, where the grid resolution is different in CRASH-AMR, Section 3.4.1 discusses the results. We have also set up test cases where the resolution at the base grid is different, but that at the finest refinement level is the same, Section 3.4.2 discusses these results.

3.4.1 Test 2 with different grid resolutions

In this section we discuss the effect of higher grid resolutions on the $x_{\rm HII}$, $x_{\rm HeII}$, $x_{\rm HeIII}$ and T profiles. We ran a test with the set-up same as that of Test 2b, but the ICs are set on different grid resolutions.

Figure 3.12 shows the spherical averages of $x_{\rm HII}$, $x_{\rm HI}$, T, $x_{\rm HeII}$ and $x_{\rm HeIII}$ from simulations with grids of 64^3, 128^3, 256^3 and 512^3 resolution, with no refinement. The bottom sub-panels show the Δ, where R_{ref} is the CRASH-AMR results with respect to the highest resolution, i.e. 512^3, and R_i are the results with other resolutions. For clarity we show Δ only for $x_{\rm HII}$ and

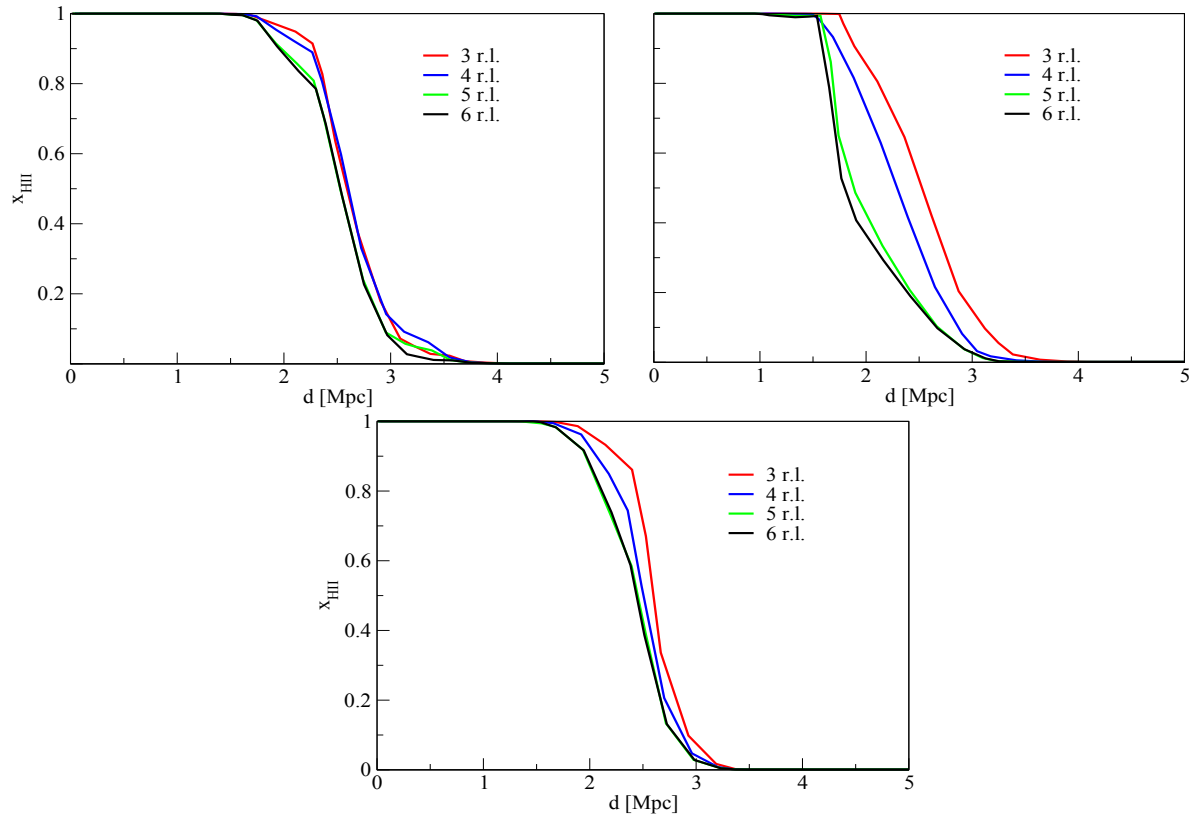

Figure 3.11 LOSs in random directions when moving away from a point source, for Test 2c at time $t = 500$ Myr. The colors refer to a test case with three (red), four (blue), five (green) and six (black) r.l.

x_{HeII} in the respective panels. We find that the results agree well with the 512^3 resolution case, with a maximum of 10 % difference for all the four profiles. Also note that the difference between the 256^3 and 512^3 results is the minimum (green line - bottom panels) for all the four profiles. This is as expected since with increasing resolution the calculation of the spherical average is more accurate and hence the difference should be minimum. As mentioned in 3.3.1.2, the spikes that we see in the profiles are numerical artefacts due to code optimization and not due to the interfacing with `CHOMBO`.

3.4.2 Test 2 with different base grid resolution and refinement levels

We then turned our attention to running test cases where the resolution at the base grid is different, but that at the finest AMR level is the same. Note that in these test cases we have completely refined the grid, although this is not usually the case with AMR codes. This sets the same grid resolution at the highest refinement level, and hence the results of the RT simulation should also match. We set up test cases with the following grid properties:

1. Base grid resolution 64^3, three levels of refinement.
2. Base grid resolution 128^3, two levels of refinement.

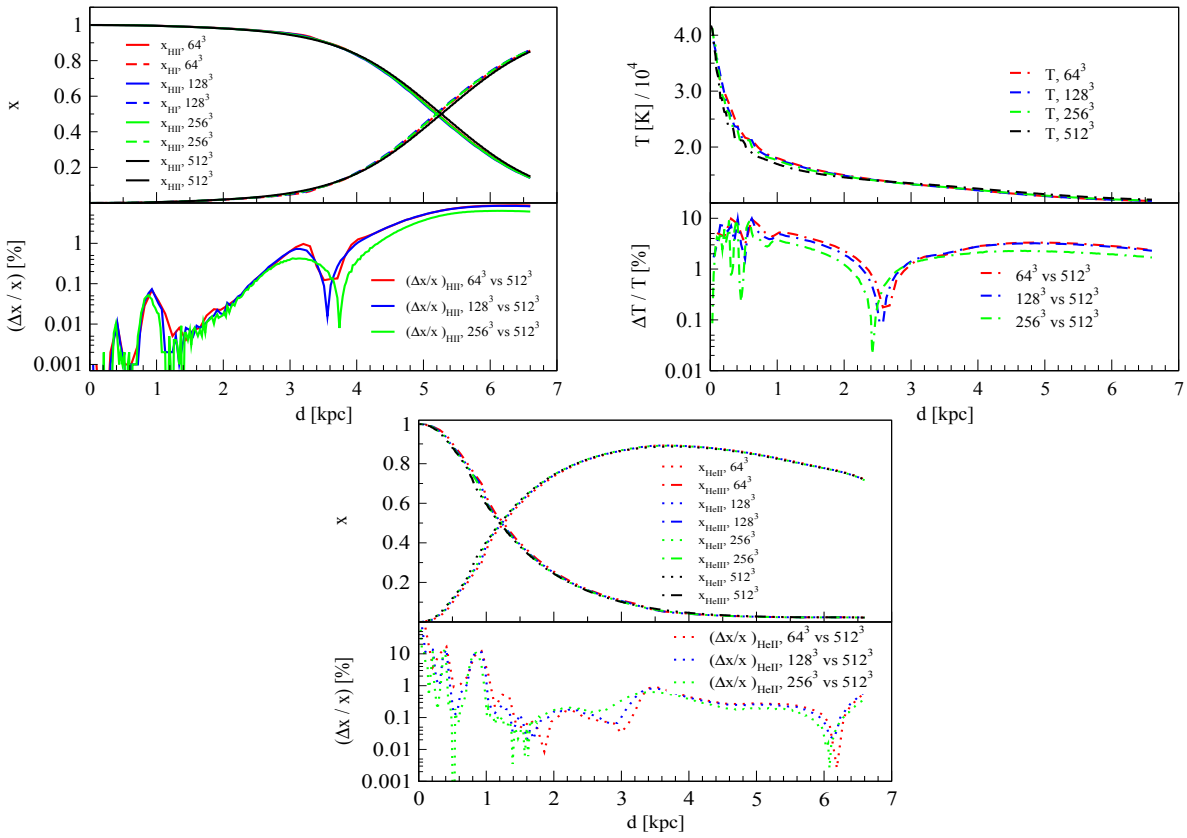

Figure 3.12 Spherically-averaged profiles at time $t = 500$ Myr for Test 2b. The colors refer to CRASH-AMR with different base grid resolutions, 64^3 (red), 128^3 (blue), 256^3 (green) and 512^3 (black). **Top:** Profiles of $x_{\rm HII}$ (solid lines) and $x_{\rm HI}$ (dashed lines). **Middle:** Profile of T (dash-dot lines). **Bottom:** Profiles of $x_{\rm HeII}$ (dotted lines) and $x_{\rm HeIII}$ (dot-dash lines). The bottom sub-panels show Δ between the corresponding values at different resolutions when compared to 512^3.

3. Base grid resolution 256^3, one refinement level.

4. Base grid resolution 512^3, no refinement.

Figure 3.13 shows the spherical averages of $x_{\rm HII}$, $x_{\rm HI}$, T, $x_{\rm HeII}$ and $x_{\rm HeIII}$ from simulations with base grid resolution of 64^3 with three refinement levels, 128^3 with two refinement levels, 256^3 with one refinement level and 512^3 with no refinement. The bottom sub-panels show the Δ, calculated as mentioned in Section 3.4.1. For clarity we show Δ only for $x_{\rm HII}$ and $x_{\rm HeII}$ in the respective panels. The grid resolution at the highest refinement level in each case is 512^3. Since the grids are completely refined, we have calculated the spherical average at the highest resolution for all cases, i.e. at 3rd refinement level for 64^3, 2nd refinement level for 128^3 and so on, so that they correspond to 512^3. The Δ between the different grid resolutions is zero which is as expected since the values of $x_{\rm HII}$ at the highest resolution, i.e. 512^3 should be the same for all cases.

The modified ray-tracing algorithm in CRASH-AMR now has to trace a ray across different refinement levels. At each cell crossing the code has to check for a new cell at the same or a

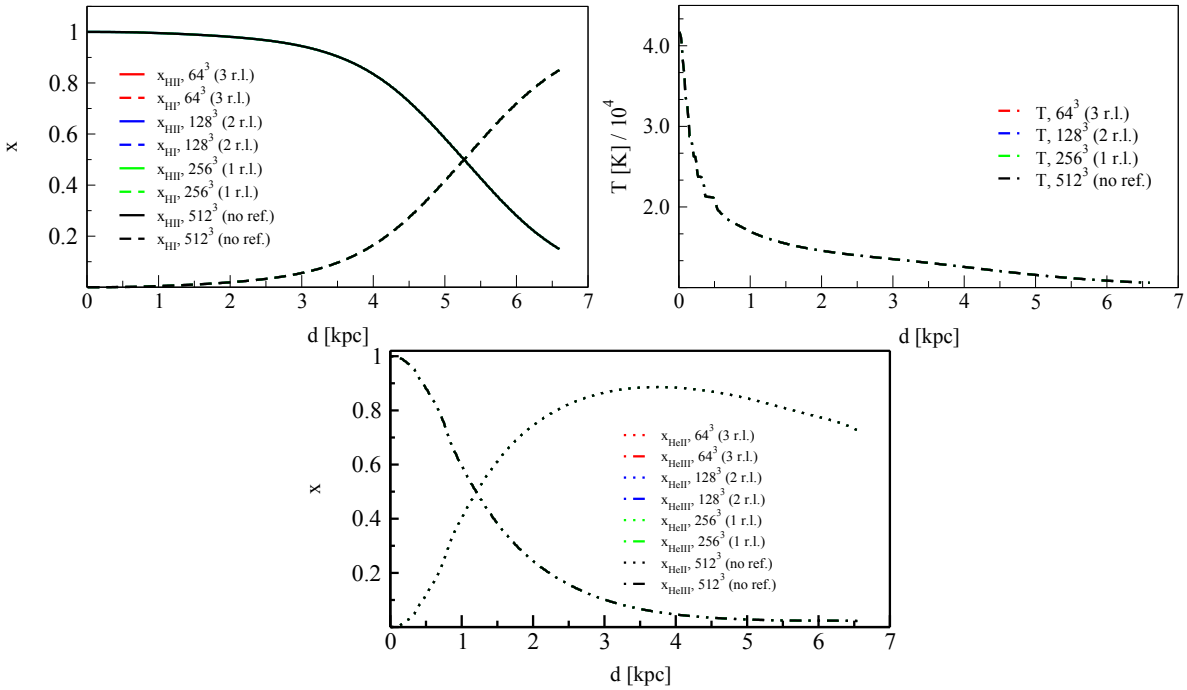

Figure 3.13 Spherically-averaged profiles at time $t = 500$ Myr for Test 2b. The colors refer to CRASH-AMR with different base grid resolutions, 64^3 (four AMR levels - red), 128^3 (three AMR levels - blue), 256^3 (two AMR levels - green) and 512^3 (no refinement - black). **Top:** Profiles of $x_{\rm HII}$ (solid lines) and $x_{\rm HI}$ (dashed lines). **Middle:** Profile of T (dash-dot lines). **Bottom:** Profiles of $x_{\rm HeII}$ (dotted lines) and $x_{\rm HeIII}$ (dot-dash lines).

different refinement level. This could have a negative impact on the run time performance of the code. We set up some test cases to verify is this was the case, and we discuss them in the following Section.

3.5 Run time performance

In Section 3.2 we had mentioned that there was no run time overhead of using CHOMBO based AMR grids in CRASH, but only the additional time to search for the new cell in the multiple refinement levels that the ray passes through. Even though this search adds to the total run time in the RT simulations done using AMR grids, its effect is small when compared to doing the RT simulation on a uniform high resolution grid. To prove that this is so, we ran some test cases comparing the run times between grids at uniform resolution and AMR grids from the CHARM simulation. Using CRASH-AMR to run the RT simulation on a uniform grid is the same as using CRASH3 instead. Our intention is to demonstrate that CRASH-AMR is able to provide results with the same accuracy as CRASH3, with much less computational efforts.

We have used a maximum grid resolution of 512^3 for our test cases. At this stage CRASH-AMR is still a serial code; it requires at least eleven, double precision, variables for its RT simulation, i.e. ~ 12 GB to store the data in a 512^3 resolution grid. Hence, for the serial version, running

Table 3.1. Run times (in hours) for CRASH-AMR simulations, similar to Test 1, along with the corresponding volume averaged $x_{\rm HII}$ at time $t = 500$ Myr.

Test case	Grid resolution	Run time (hours)	$x_{\rm HII}$
U128	Uniform grid (UG), 128^3	1.60	0.181
U256	UG, 256^3	3.55	0.185
U512	UG, 512^3	8.32	0.183
R128	Refined grid (RG), 128^3	1.58	0.180
R256	RG, 256^3	2.52	0.184
R512	RG, 512^3	3.40	0.181

the RT simulation on a uniform grid at larger resolutions is not possible.

3.5.1 Set up with single point source

The set-up is similar to Test 1, with the ionised region from a single point source expanding into a realistic density field. The source is placed at the highest density peak and is assumed to be steady, with $\dot{N}_{\gamma,ref} = 5 \cdot 10^{54}$ photons s^{-1}, a monochromatic spectrum of $h\nu = 13.6$ eV. The source emits $N_p = 2 \cdot 10^8$ photon packets, this value ensures that we reach convergence by the end of simulation (see Section 3.1.1.1 for a discussion). The gas is assumed to contain ony H with an initial ionisation fraction $x_{\rm HII}$ set to $1.2 \cdot 10^{-3}$. The gas temperature, T, is initially set to 100 K and is kept constant throughout the simulation. The simulation time is $t_{sim} = 500$ Myr, starting at redshift $z = 0.1$. We output the results at intermediate times t = 10, 20, 100, 200 and 500 Myr.

To compare the run times we have created uniform resolution grids from the AMR refined grids used in Test 3 by interpolating the coarse data onto the finer levels. We use grids of resolution 128^3, 256^3 and 512^3 and compare the run times to that of grids with base resolution of 64^3 with 1, 2 and 3 refinement levels respectively. We would like to point out that there are other factors that also impact the run time of a CRASH simulation, for example the number of point sources, presence of He with T evolution and number of photon packets. However, we do not consider them here and focus instead on the performance of the code for a simple test case.

Table 3.1 shows the run times on grids at uniform resolution of 128^3 (*U128*), 256^3 (*U256*) and 512^3 (*U512*). We compare these with the run times on grids with a base resolution of 64^3 with 1 (*R128*) , 2 (*R256*) and 3 (*R512*) refinement levels respectively. The corresponding volume averaged $x_{\rm HII}$ at time $t = 500$ Myr are also shown.

We compare test cases *U128* with *R128*, *U256* with *R256* and *U512* with *R512*. For cases *U128* and *R128*, the difference in run times is only 1.25 %, but as we go to higher resolutions for example cases *U512* and *R512*, the difference in run times is as high as 59 %. The corresponding difference in volume averaged $x_{\rm HII}$ is only 0.5 % and 1.1 %.

Table 3.2. Run times (in hours) for CRASH-AMR simulations, similar to Test 3a, along with the corresponding volume averaged $x_{\rm HII}$ at time $t = 500$ Myr.

Test case	Grid resolution	Run time (hours)	$x_{\rm HII}$
U512	UG, 512^3	11.4	0.0501
R512	RG, 512^3	6.97	0.0494

Figure 3.14 compares two random LOSes, moving away from the source, for the case with uniform and refined grid at different resolutions. The top panel compares the LOSes from *U128* and *R128* test cases, the middle panel compares *U256* and *R256* and the bottom panel compares *U512* and *R512*. We find that the extent of the fully ionised H II region for the uniform and AMR grid cases is the same for all the test cases. The partially ionised regions show the largest differences; the size of the I-front is smaller for all the test cases where we use refined grids when compared to the uniform grid case, with a maximum difference of 2 %.

3.5.2 Set up with multiple point sources

We look at another test case, similar to Test 3a, to confirm that CRASH-AMR provides accurate results with shorter run times, when compared to CRASH3. We use a grid at a resolution of 512^3 and compare the run time to that of a grid with a base resolution of 64^3 with 3 refinement levels, see Section 3.3.3.1 for details on the set up.

Table 3.2 shows the run times on grids at uniform resolution of 512^3 (*U512*) compared to the run time on grids with a base resolution of 64^3 with 3 (*R512*) refinement levels. The corresponding volume averaged $x_{\rm HII}$ at time $t = 500$ Myr are also shown. We find that the difference in run times between the two cases is as high as 38 %. The corresponding difference in volume averaged $x_{\rm HII}$ is only 1.3 %.

Figure 3.15 compares three random LOSes, moving away from the most luminous point source. We show the same LOS, as plotted in Figure 3.9, for *R512* and compare the results with that of *U512*. Here as well we find that the size and extent of the ionised H II region formed in the test case with refined grids matches well with that of the uniform grid. We find a maximum difference of 2 % in the size of the I-front.

From the two test cases above it is clear that, quantitatively, CRASH-AMR is able to provide results that are as accurate as CRASH3 but with much less run times. We have been able to represent the regions of interest at a resolution increased by a factor of 64 and not lose any accuracy in the results. Our interface with CHOMBO and the ray-tracing algorithm to search for the new cell across different refinement levels during ray-tracing is highly efficient and does not have a negative impact on the run time. CRASH-AMR is an improvement over CRASH3 both in terms of run time and memory consumption; this will allow us to carry out RT simulations at a much higher resolution than was possible earlier.

3.6 Conclusion

We have introduced CRASH-AMR, a new version of the cosmological radiative transfer code CRASH, enabled to run RT simulations on AMR grids. After an exhaustive introduction of the code development we have shown the results of many tests both with the simplified set-up prescribed in RTCCP and a realistic hydrodynamic simulation with AMR refinement. All the tests show good agreement with the latest release of CRASH, confirming the correct introduction of a more accurate and alternative geometry representation of the gas distribution in the cosmological domain in which the RT is performed. Few differences found in our tests show that small discrepancies are just due to the introduction of a higher grid resolution of the refined levels or are simply ascribable to numerical artifacts introduced by averaging operations.

Once applied to a more realistic density field, we find a sharp difference in the ionised region patterns formed at different AMR resolutions because of the more accurate treatment of the gas optical depth and cooling function. Consequently, the ionisation fractions of the H, He species are calculated with greater accuracy at higher resolutions and the global process of ionisation proceeds with accuracy in reproducing the bubble growth and the escape of ionising radiation from high-density regions in which star formation is generally embedded.

Regarding the many advantages of CRASH-AMR, we note that the new code is able to perform RT simulations in high density regions with the resolution increased by a factor of 64, without experiencing serious memory limitations. It has been proven to be able to manage the entire large set of information usually handled by the original version and to account for a large set of physical processes. Such a high resolution would be unmanageable from the storage point of view in a single compute core without this new version. We also find that when compared to running on uniform, high resolution grids, we get up to 59 % reduction in run times when the same set up is applied on smarter AMR grids; hence CRASH-AMR provides an advantage both in terms of memory consumption and run time performances when compared to the standard version of CRASH.

The adaptivity of the grids used in CRASH-AMR is decided by the hydro code and CRASH-AMR is only used in a post-processing mode. However, the capability of adaptively refining the grid within the RT simulation is of importance in certain situations, for example, studying the contribution of quasars in IGM reionisation. As a start, we would like to use this adaptivity feature for a test case with a single point source such as a quasar. The other application of CRASH-AMR, that is of more importance, is the setting up of cosmological simulations on larger volumes, with higher resolutions, and hundreds, if not thousands of point sources. So we first look at parallelising CRASH-AMR using distributed memory parallelism. Then we look at the feasibility of implementing adaptive RT in CRASH-AMR.

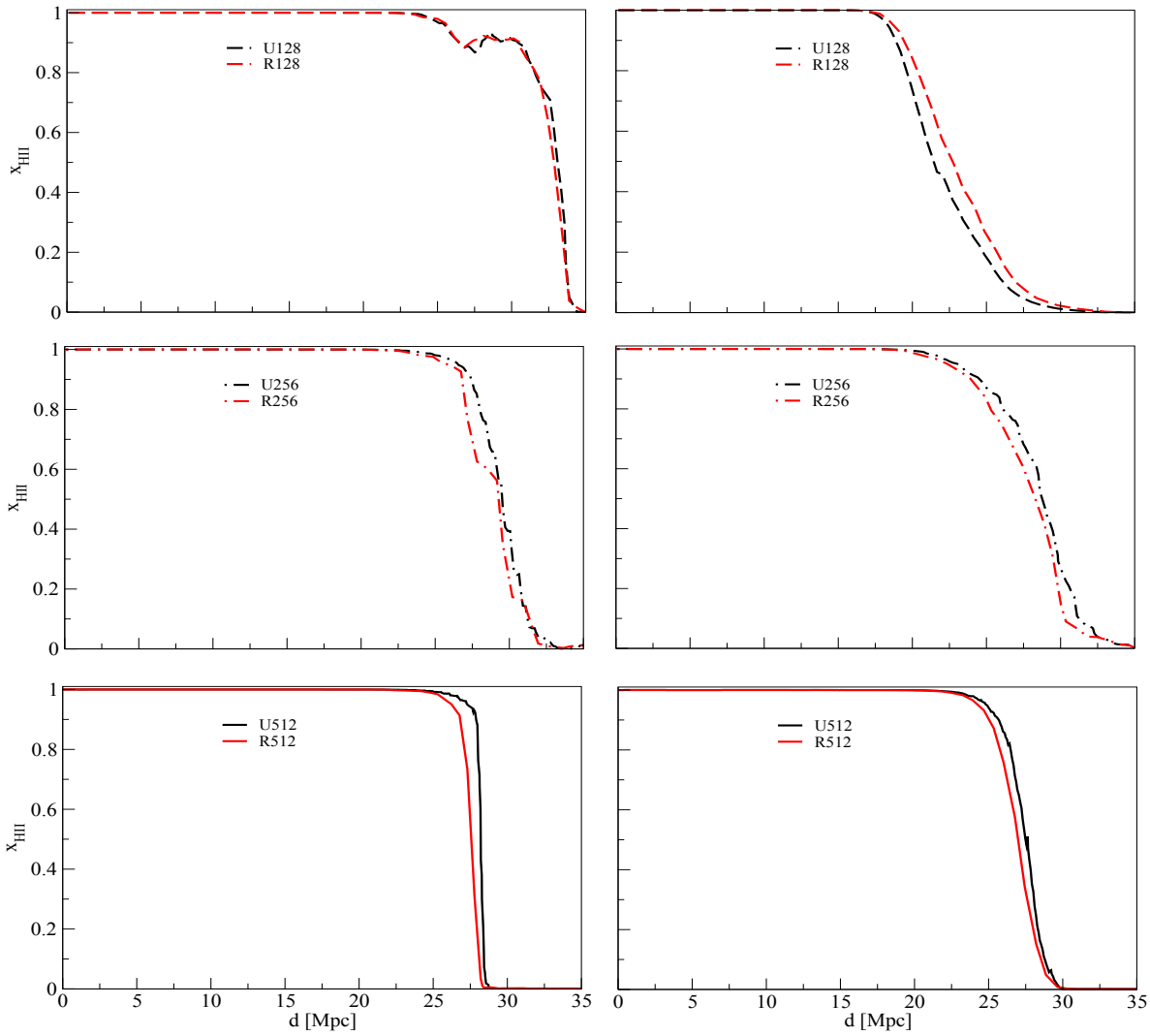

Figure 3.14 LOSes in random directions when moving away from a point source, for Test 1 at time $t = 500$ Myr. The colors refer to `CRASH-AMR` with a uniform grid (black) and with r.l. (red). **Top:** U128 and R128 (dashed lines). **Middle:** U256 and R256 (dot-dash lines). **Bottom:** U512 and R512 (solid lines).

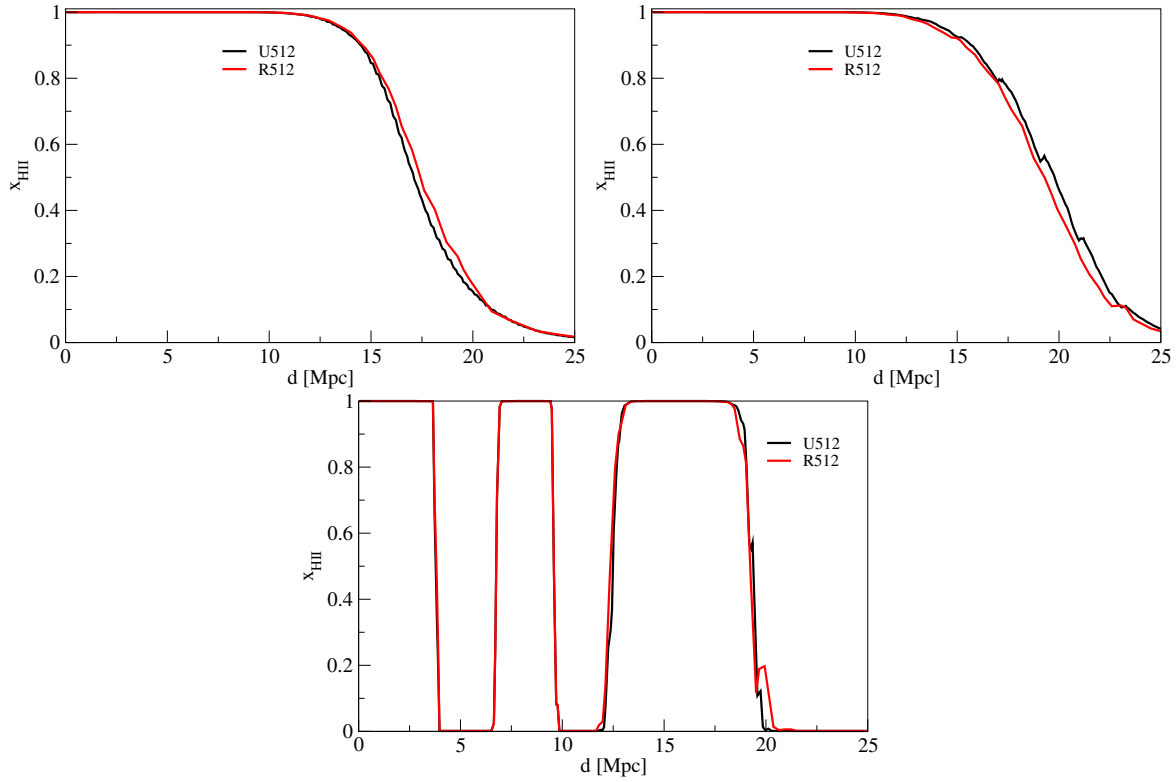

Figure 3.15 LOSes in random directions when moving away from a point source, for Test 3a at time $t = 500$ Myr. The colors refer to CRASH-AMR with a uniform grid (black) and with 3 r.l. (red).

Chapter 4

Parallelisation of Radiative Transfer in CRASH

In Chapter 3 we spoke about the development of a new code CRASH-AMR that can do RT simulations on static, nested grids that are the outputs of a hydro simulation. We use the term static to refer to the fact that the adaptivity of these grids is decided by the hydro code, while the RT simulation does not carry out any refinements. CRASH-AMR is a serial code, and our next effort has been towards parallelising it using distributed memory parallelism. Even though a parallel version of the CRASH code exists (PCRASH [144]), we have used CRASH3 as our base version for further developments. Our motivation to do so is due to the fact that the software architecture of CRASH3 makes it amenable for addition of further physics modules or other algorithmic improvements. Also, when compared to PCRASH, it has the modules necessary to include metals [73]. Even though we do not include metals in CRASH-AMR now, it can be done easily in the future as the initial framework to do so already exists.

Parallelising CRASH-AMR will give us the advantage of being able to run RT simulations on larger box sizes with more point sources. Also, we could look at the possibility of carrying out RT simulations with cosmological set ups, by processing multiple snapshots. We begin by discussing some of the techniques used to parallelise SAMR codes and the load balancing techniques they adopt in Section 4.1. Next we look at the scheme we have adopted to parallelise CRASH-AMR in Section 4.2. Finally, we discuss a number of tests that have been done to study the performance our parallel code in Section 4.3.

4.1 Parallelising SAMR codes

Parallelisation of SAMR codes presents a unique set of challenges, when compared to codes that use uniform grids, in terms of data-distribution and domain decomposition across cores. The aim, as always, is to minimise communication and synchronisation events between the various participating cores, while ensuring an optimal load balance. The complexity with SAMR codes arises due to the fact that as the simulation progresses, the grid hierarchy can

change due to addition or deletion of new refinement levels [167]. This requires new strategies for an efficient parallel implementation.

The first step towards this parallelisation is to have suitable data structures to represent the grid hierarchy. This allows the solution to be separated into different components that corresponds to the grid hierarchy, the associated functions and components that relate the two. In Section 2.2.1.2.5 we had discussed the different data structures required to implement SAMR. The methodology adopted in CHOMBO, discussed in Section 2.2.6.2, provides the necessary framework for parallelisation. The library provides classes, such as the *DataIterator* and *LayoutIterator*, that clearly distinguish between data that is local or non-local to a core. Separate classes provide the functionality required to solve the PDEs associated to the problem under consideration. The presence of independent grids at each level allows concurrent operations to be done, thereby exploiting the inherent parallelism of SAMR.

With a well-defined data structure in place, one can now look at the various aspects of domain decomposition required for adaptive codes. The key requirement for domain decomposition schemes is to ensure [168]

(a) A logical consistency of the grid hierarchy across cores, in terms of information relating to which core owns a particular grid [142].

(b) Spatial locality of grids which is necessary to reduce the number of communications between cores. This communication can be classified as "intra-grid" and "inter-grid". Intra-grid communications are typically done to copy the boundary data between grids that lie on the same level, these can be overlapped with computations done in the interior parts of the grid. Inter-grid communications take place between grids at different levels of refinement, involving interpolation and averaging operations among the child and parent grids.

(c) Proper load balance across cores. The hierarchy in AMR consists of grids at different resolutions, thus resulting in different computational loads associated with each grid. An efficient domain decomposition scheme provides a balanced load distribution across processors thereby reducing idle time among cores. The load balancing (LB) scheme employed can be either static or dynamic depending on how often the grid hierarchy changes.

We now discuss points a, b and c in more detail.

The data structure used to maintain the AMR hierarchy reflects the relationship between the grids at different refinement levels. This can change during a simulation, due to addition and deletion of grids. It is then necessary that the information pertaining to the neighbors, child or parent patches are updated for each grid at all refinement levels. This can become a bottleneck if all the cores need to replicate the entire hierarchy requiring synchronisation events to get the updated patch information from other cores. Some AMR codes, such as CHOMBO, replicate the metadata containing the entire grid hierarchy across all the cores. The AstroBear library, on the other hand, makes use of a distributed tree, and each core stores only the part of the AMR hierarchy necessary to carry out computations and communications [133].

Preserving the spatial locality of grids in SAMR depends on the domain partitioner used, these can be broadly classified as *domain-based* (DBP), *patch-based* (PBP) and *hybrid* (HP) methods [92]. The DBP method [143, 177, 187] works by partitioning the whole physical domain rather than the grids. This results in grids across all levels being assigned to the same core. This can be further optimised by using Space-filling curves (SFC) to order the grids in a one-dimensional list which is then cut according to a threshold value for the load [52, 142]. The DBP method has an advantage of reduced communication volumes. The PBP method, on the other hand, works by distributing the grids at each level independently. This in theory provides perfect load balance but can result in more communication among cores during synchronisation events [104, 151]. The HP method works by combining the advantages of the DBP and PBP methods [105, 179, 187].

The DBP, PBP and HP methods discussed provide different strategies for balancing the load across cores. These can be used as both static and dynamic LB schemes. In a static LB scenario, the grids are distributed at the beginning of the simulation. In a dynamic LB scenario, the loads needs to be redistributed due to the addition of new grids. Here, the advantage of the PBP method becomes clear. Distributing the new patches introduced by SAMR can be done fairly easily in the PBP method as it works at each level independently. The new level of refinement could however add to the inter-grid communication and synchronisation costs. The DBP method, when used for dynamic LB, requires a complete repartitioning of the domain across all levels to avoid inhomogeneous work loads [152]. It was also found that the DBP method works well only for grid hierarchies with up to three levels of refinement and performed poorly when higher refinement levels were present [178]. The use of inverse SFC methods have been shown to be a promising alternative instead [143, 176]. Another method suggested in [179] for dynamic LB involves the use of a meta-partitioner that can select from a list of partitioning schemes available and depending on the current state of the application can redistribute the load. A similar technique suggested in [91] uses a database consisting of application states and their associated performance measurements for different partitioning algorithms. The scheme that gave the best performance for an application state closely matching the current state is then selected.

Now that we have a general overview of the different aspects involved in parallelising SAMR codes, we look at the techniques used by some other ray-tracing codes for parallelisation. In particular, we look at ray-tracing codes that work on refined grids.

4.1.1 Parallel ray-tracing codes

Ray-tracing codes like `CRASH` are suitable for parallelisation since each core can work independently by emitting the rays from the point-sources in its domain and follow them till they exit the domain. However, their performance depends on careful considerations that need to be applied for decomposing the domain among cores. Additionally, the ray-tracing algorithm also has a huge impact on the efficiency of the code. We provide some examples of such parallel ray-tracing codes, and mention the techniques they adopt for parallelisation.

The `PCRASH` code [144], based on a previous version of `CRASH`, carried out parallel RT simulations and demonstrated good scaling up to 128 cores. The code made of uniform grids and

employed a static LB scheme. The domain was decomposed among cores by using the Hilbert SFC [8], and the load on each cell was calculated using criteria specific to RT simulations.

The adaptive ray-tracing method in MHD code ENZO uses a SAMR scheme for its hydrodynamic simulations. The RT module of the code is based on an adaptive ray-tracing technique, which has been parallelised using MPI [198]. ENZO stores the complete grid hierarchy in each core and utilizes a "non-blocking" scheme where the packets are not propagated according to the time at which they were emitted, but according to when they were received by a core. This removes the need to synchronise between the different iterations. The code makes use of a BSAMR framework, which has some differences to the PBAMR framework used in CRASH-AMR. This changes the way ray-tracing can be done in ENZO since a one-to-one relationship between a child and parent grids exists, which is absent in CRASH-AMR. The load on each box is set to the number of cells within the box; both static and dynamic LB schemes are available which make use of the DBP, PBP and Hilbert curves to distribute the load. The code shows good scaling up to 512 cores.

The FLASH code makes use of a "hybrid" method to do ray-tracing [32, 154] that takes advantage of both the long and short characteristic methods, described in Section 1.3.1, to calculate the optical depth or gas column density. Similar to CRASH-AMR the boxes at each level are assigned a *globalID* and here as well all the patches are known to all the cores. It also implements the BSAMR framework, similar to that of ENZO, where grids at different levels do not overlap but are completely contained within the parent level below. The static LB scheme adopted ensures that the AMR hierarchy is equally split among all cores by assigning an equal number of "leaf blocks", i.e. grids that are not covered by any other grid, to each core. The tests done with the code scale up to 350 cores.

We also mention RADAMESH, an adaptive MCRT code that utilizes the BSAMR framework for its adaptivity. Differently to the codes mentioned above, it has been parallelised using OpenMPI and has been run on eight cores.

We have given some examples of ray-tracing codes that have been parallelised using different techniques. Most of them make use of the BSAMR framework, and store the complete AMR hierarchy on each core. Similar to these codes, CRASH-AMR also has to propagate the ray through multiple refinement levels. Also, the problem is exacerbated by the fact that the rays travel in random directions and the parts of the domain they will or should cover cannot be decided a-priori. Depending on the geometry of the density field, the rays can travel long or short distances. In our case we have yet another additional complexity that the PBAMR framework does not have a tree structure where a one-to-one relationship between the child and parent box exists. We need to use some searches to find the right parent box. All these factors put together complicate ray propagation in CRASH-AMR and consequently its parallelisation as well. Nevertheless, we take a look at some techniques that can be applied to parallelise CRASH-AMR and demonstrate our initial attempts in this direction.

4.2 Parallelising CRASH-AMR

Parallelising CRASH-AMR has required a number of changes, in terms of the load balance and domain decomposition across cores, the interface with CHOMBO to get the grid information back

to CRASH-AMR and the way the RT simulation itself is done. We mention these separately to give a complete picture of the methodology adopted to parallelise CRASH-AMR. Henceforth, we refer to the parallel version as PCRASH-AMR to distinguish it from the serial version CRASH-AMR.

The CHOMBO library provides thread level parallelism using OpenMP and process level parallelism using MPI. We use the relevant features in CHOMBO to parallelise PCRASH-AMR using MPI. The library provides a number of routines to facilitate the data management and distribution across cores. It also provides relevant classes that can access the whole grid hierarchy now lying across cores. We will mention them as and when they are used. Calls to initialise and finalize MPI need to be taken care of by the user. The default communicator is *Chombo_MPI::comm*, instead of *MPI_COMM_WORLD*, and is converted to a Fortran integer for use within PCRASH-AMR.

We discuss in detail the parallelisation scheme in PCRASH-AMR, and begin by looking at load balancing technique used in the code which impact the other two changes we mentioned above.

4.2.1 Load balancing

For the case of the RT simulation on a uniform grid, the load on each core depends on

1. Number of point sources that are in the core domain: the photon-packets originating from these sources have to be propagated and, if necessary, communicated.

2. Distance of a cell in a box from a point source: the closer the cell is to the source, the more photon-packets cross it. This contributes to the total load in a box assigned to a core.

3. The luminosity of a source: this can impact how far the photon-packets are able to travel before they are absorbed by the medium. The further the packets are able to travel, the larger the number of communications required to propagate the ray. Also, initially the gas around the point source might be dense enough to prevent any photons from escaping. As the simulation proceeds, the gas gets ionised, resulting in the photon-packets traveling further before they get completely absorbed. This results in a scenario where the packets emitted initially do not cross many cells, but at later times travel further and require more MPI communications to be propagated.

In the case of RT simulations on multiple grid levels, we need to take into account an additional factor. If a source lies in a box belonging to a core, P_0, then there is an additional expense attached to the box if its neighbor(s), child(ren) or parent(s) lie in other cores. Every time the ray exits the box and moves to another box, we need to communicate the ray. One could in principle use a DBP approach and assign the boxes that cover or are covered by other boxes to same core. Although this strategy reduces the communication involved in the RT code, it is rather inefficient for our purposes as the resulting load distribution might assign a large number of boxes to one core. Take, for example, the refined grids obtained from the CHARM hydro simulation as shown in Figure 4.1. We show a 2D image with the outlines of the boxes at coarser (red boxes) and finer (green boxes) resolutions. The domain has been split among four cores as per the criteria mentioned. One can immediately see that with this

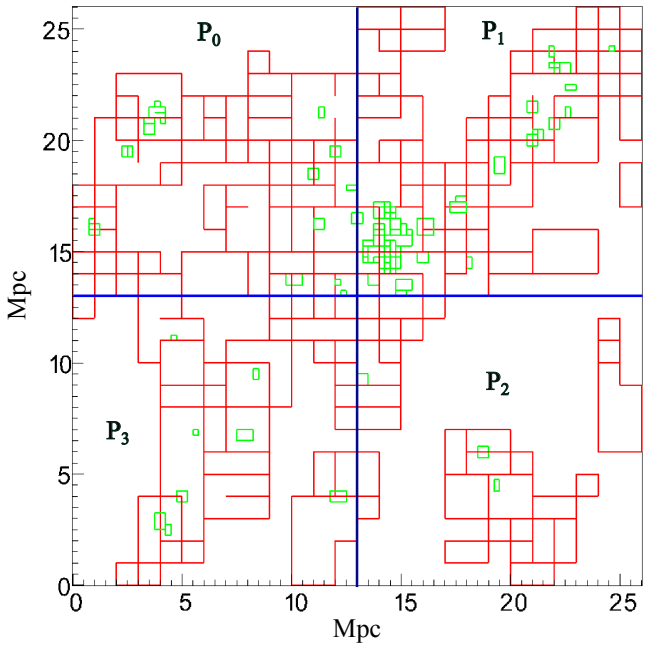

Figure 4.1 2D image showing the refinements around high density regions for the simulation outputs from the CHARM code, and the domain decomposed among four cores (P_0, P_1, P_2 and P_3). The red boxes show the grids at coarser resolutions, the green boxes are at higher resolutions.

method core P_1 has a larger number of green boxes, where point sources will be located and distributing the load this way will not be conducive to the code performance. Also, CHOMBO as of now does not provide a DBP approach. We have instead used the PBP approach for partitioning the domain which we discuss this further in the context of PCRASH-AMR. Note that we do these simulations in a post-processing mode, i.e. the grid hierarchy in the code does not change during the simulation, so we only consider a static LB scheme as of now.

To calculate the load on each core, we provide the user two different LB routines to choose from. This selection can be done during the set up of the simulation ICs. The first option calls the default LB routine of CHOMBO. This sets the number of cells in each box as the load on each box, similar to the scheme followed in [198]. Next, an array containing the load for each box, $l_0, l_1, .., l_N$, for the N boxes at each refinement level is specified. A "modified Knapsack algorithm" is then used to balance the load. This starts by calculating an average load on each core and assigning the first box to a core with MPI rank 0. For each subsequent assignments, it calculates a dynamic goal based on the remaining load. This ensures that the last core does not end up with an abnormally large workload. To improve locality, boxes with equal number of cells can be swapped among cores.

The second option calls an alternate implementation of the LB routine in CHOMBO which accepts an array of loads calculated with user-defined criteria. In our case, we set the load on each box to be the "ray density" as defined in [144] . In an optically thin medium, the "ray density" (RD) in each cell is proportional to the number of rays that passes through the cell and can be defined as

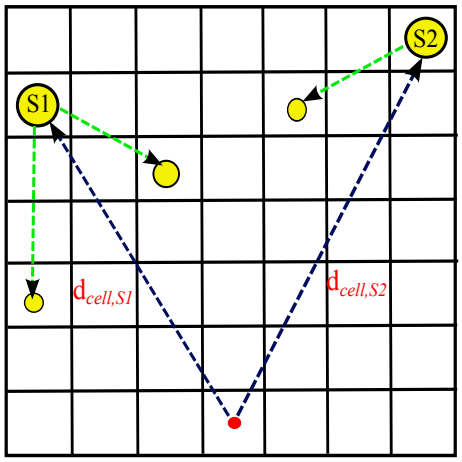

Figure 4.2 Image showing five point sources with different luminosities in a grid. Sources $S1$ and $S2$ have a luminosity higher than that of the other three sources. A tree structure is formed with $S1$ and $S2$ as the parents, and the sources near them as child(ren). In the image, parents and child(ren) are connected by green arrows. The ray density from sources $S1$ and $S2$ and their child(ren) is calculated for the cell containing the red dot. See text for more details.

$$RD_{cell} = \sum_{sources} d_{cell,source} \tag{4.1}$$

where $d_{cell,source}$ is the distance between the point source and the cell for which the load is being calculated. RD_{cell} is calculated by summing the load on the cell from all the sources. For a uniform grid, since all cells are at the same refinement level, this can be calculated easily. In PCRASH-AMR we need to make some improvement as follows.

Once we have loaded the files that specify the source coordinates, we do the following:

1. We first sort the point sources in a decreasing order of luminosity, for convenience. Then starting from the most luminous source, we build a tree structure linking all the other sources (child) that lie within a certain radius from the more luminous source (parent). The radius is a user specified variable provided when the ICs are set up in the RT simulation. Then we find the next most luminous source which has not been added to the tree, and we locate its child sources. This is repeated till all the sources have been added to the tree.

2. Next, we mask out all cells at coarser levels that are covered by finer cells. As the ray will not propagate through these cells during the RT simulation (in fact the ray will only cross the cells at the most refined level), the load on them can be set to 0.

3. We loop through all remaining cells, and for each one we find RD_{cell} as given in Eqn. 4.1, summing up the values for all parent sources in the tree. If the cell lies in a refinement level different than the one of the source, we scale up/down its coordinates to the source level and then calculate the RD. The cell RD can then be written as:

$$RD_{cell} = \sum_{psources} d_{cell,psource}(1 + N_{child,psource}) \tag{4.2}$$

where $N_{child,psource}$ is the number of children each parent source *psource* has, and the sum extends only over the parent sources.

Figure 4.2 shows a sample grid with five point sources depicted as yellow circles, the size of which indicates the source luminosity. Sources $S1$ and $S2$ have a luminosity higher than the one of the nearby sources. The dashed green lines connect $S1$ and $S2$ to the less luminous sources that lie within a radius of three cells in each direction. The red dot indicates the cell for which we want to calculate the load. We determine RD due to sources $S1$ and $S2$ including the number of child(ren) in their respective trees, given by the equation

$$RD_{cell} = d_{cell,S1}(1 + N_{child,S1}) + d_{cell,S2}(1 + N_{child,S2}) \tag{4.3}$$

where $N_{child,S1} = 2$ and $N_{child,S2} = 1$.

4. Once the load on each cell has been determined, this is summed up to give the load, l_b, for each box at each refinement level.

We can now specify an array containing the load for each box, $l_0, l_1, .., l_N$ for the N boxes at each refinement level. This is passed as a parameter to the LB routine which balances the load at each level separately by first sorting the loads in increasing order. Then in decreasing order, starting with core P_0, the loads are assigned to the next core with the minimum load. If any core has a large deviation from the average load, then it is swapped with another core. The swapping is done only if it improves the efficiency, i.e. the ratio of minimum to maximum load, of the assignments.

At the end of the LB routine, we have a new distribution of boxes which can be used to create a new *DisjointBoxLayout*. The data stored in *FArrayBox*, backed up prior to the load distribution, is copied back to the new layout. Note that this copy involves calls to MPI routines to exchange the data contained in a core with another core that now owns the relevant box.

This concludes our discussion on the LB method used in PCRASH-AMR, we next look at the changes done to the interface between the two codes.

4.2.2 Changes to PCRASH-AMR and CHOMBO interface

The interface between CRASH and CHOMBO sets up the relevant information from CHOMBO required to do ray tracing in CRASH. For the case of CRASH-AMR this only involved setting up the grid hierarchy and getting back pointers to the data in each box. For the case of PCRASH-AMR, the interface has to be set up differently.

4.2.2.1 Setting up local and global lists

Before starting a RT simulation with CRASH-AMR, the grid hierarchy needs to be built using the data coming from CHOMBO. The interface between the two codes determines, for each box at each refinement level, its neighbor(s), child(ren) and parent(s). Then the physical data stored in all the boxes is also passed back to CRASH-AMR. In PCRASH-AMR, building the grid hierarchy needs to be done differently. Also the amount of physical data that each core now needs to store is less than CRASH-AMR. We explain it in detail below.

To do RT on multiple cores, we need to decompose the domain across them. In Section 4.2.1 we mentioned the steps taken by the LB routine to distribute the boxes among cores. Boxes that belong to a core are said to be 'local', while those lying in other cores are said to be 'non-local'. All cores can access the properties of all boxes, i.e. the start and end coordinates, the refinement level the box belongs to etc. However, the core can access the physical data stored in a box only if it is local, via the FArrayBox class. The data in a non-local box is not accessible implicitly and needs explicit MPI calls.

During the RT simulation, it is enough if the core can access local data. To simulate radiation-matter interaction, we solve the relevant equations using the data in only one cell, belonging to a local box, at any given instance. Thus, we do not need any access to the physical data of a non-local box. If the ray travels to a box that belongs to another core, we need to find the new box and the core that contains this box. Then the ray has to be communicated to the corresponding core for further propagation. Hence, in addition to the complete local data, we need to keep a list of non-local boxes as well.

Our approach was to create two maps in each core, 'local' and 'global'. The local map contains all the boxes that are local to a core, and has a data structure similar to the one we build in 3.2.2; the global map, on the other hand, contains all those boxes that are not local to a core. Figure 4.3 shows the data representation in CHOMBO on the left-hand side. The CRASH equivalent on the right-hand side now has two maps, local and global. The local boxes, $B(0,0)$, $B(0,1)$ and $B(1,4)$ are shown with bold lines whereas the boxes in the global map $B(1,2)$ and $B(0,3)$ are shown with dotted lines. Both these maps are built across all cores, for their respective local and non-local boxes.

We now discuss how these maps are built. The *DataIterator* class in CHOMBO gives a list of all local boxes at all refinement levels. The *LayoutIterator* class, on the other hand, provides access to the properties of all boxes, both local and non-local at all levels. Each core loops through all the boxes in the *LayoutIterator* and compares its MPI rank with the MPI rank of the box containing the core. If it is non-local, we add the box properties and the corresponding MPI rank to the global map. If it is local, it is added to the local map along with its list of local neighbor(s) and parent(s). The physical data contained in the corresponding *FArrayBox* class is also returned via Fortran pointers. Additionally, for the local boxes, we determine if its child(ren) are local or not. If the child is local, we store its *globalID* as done in 3.2.2 for easier ray traversal. If not then we assign a *globalBoxID* of -1 to indicate that it should be searched in the global map. While building the maps, we ensure that the unique pair of *localID* and *globalID* of a box is the same across all cores. This makes it easier to locate the right box during ray propagation across cores.

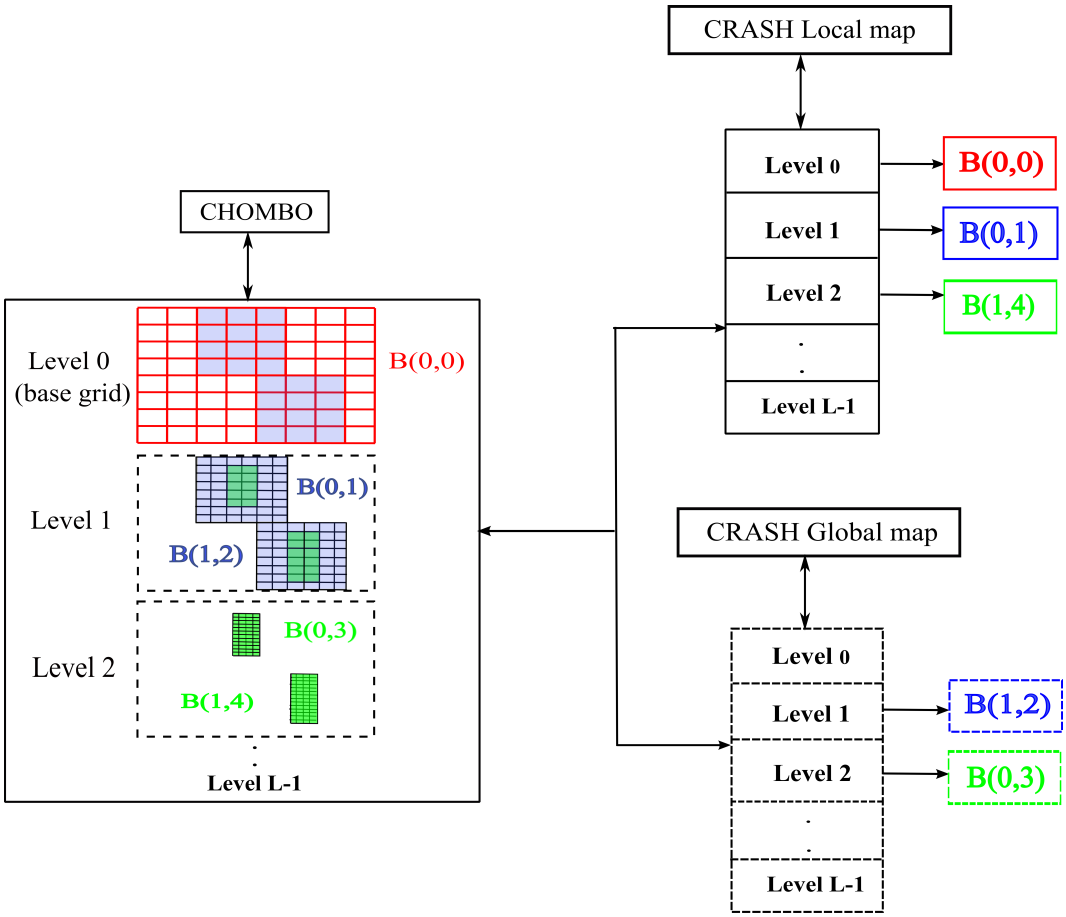

Figure 4.3 Interface between CRASH and CHOMBO, the grid hierarchy in CHOMBO being reflected through the data structure built in CRASH. The AMR grids, stored as an array of boxes in CHOMBO, are now stored in two maps, local and global. The local map contains the boxes belonging to a core (solid lines). The non-local boxes belonging to other cores are stored in the global map (dotted lines).

The global map built as mentioned above contains all the non-local boxes. This is similar to the approach followed in [32, 198] where each core stores information related to all the patches in the AMR hierarchy. In principle, for RT simulations, it is not at all necessary to store all this information since the ray will only ever travel to a surrounding box. To get a list of these boxes, we only need to store in the global map the following information

(a) Non-local neighbor(s), child(ren) and parent(s) of local boxes

(b) Neighbor(s) of non-local child(ren) and parent(s)

(c) Child(ren) and parent(s) of non-local neighbors

This in effect gives a complete list of the non-local boxes that surround the boxes local to a core. During ray-tracing, the cells crossed are either in the local boxes, or in the boxes that satisfy the criteria mentioned above.

Finding the information mentioned in (a) is simple enough and can be done as follows

1. The *NeighborIterator* class gives access to both local and non-local neighboring boxes. The properties of a non-local box can be found using the *LayoutIterator* class.

2. To find the non-local child(ren) and parent(s), we loop through the disjoint set of boxes in the levels above and below. The *coarsen*, *refine* and *isEmpty* routines in the *Box* class then help find the relevant child or parent boxes. The MPI rank of the core containing the box indicates if it is local or not.

Finding the information mentioned in (b) and (c), however, proves to be non-trivial since it requires determining the neighbors of non-local boxes, while the *NeighborIterator* class can determine the neighbors of only local boxes. We also require careful additional checks and MPI communications among cores to set up this information. For now we have created the global map with all non-local cores. Setting up this map is straightforward as the required information is readily available in CHOMBO. The downside is that it will require superfluous searches to find the right non-local box to propagate the ray through. We intend to optimise the global map, by storing only the neighboring patch information, as it is evidently more optimal. Once we have the maps set up, we next look at how these maps are used in PCRASH-AMR.

4.2.3 Changes in PCRASH-AMR

Now that we have mentioned the load balancing technique and the changes required for the interface between PCRASH-AMR and CHOMBO, we describe the changes done in PCRASH-AMR with respect to the ray-tracing algorithm.

4.2.3.1 Point sources

During the setting up of ICs for a simulation, the file containing the point source coordinates is loaded and stored as a linked list for easy traversal. In PCRASH-AMR, since the boxes are

distributed, each core only needs to propagate the sources that lie in its local boxes. So, at the beginning of the simulation, each core finds the sources that lie in its local map and updates the linked list to reflect this.

4.2.3.2 Multiple photon-packets per time step

In Section 3.1.1 we had discussed about the time resolution in CRASH-AMR, which is decided by N_s and N_p. This impacts the cell crossing times and, consequently, the equations that evaluate the radiation-matter interaction. The ray-tracing algorithm works by emitting a photon-packet from a source in a random direction, and following it till it exits the grid. In the case of multiple point sources a photon-packet is emitted successively by all the sources at time $t_j = j\ dt$, where dt is the time step as defined in Section 3.1.1. Each packet, from each source, is followed till it satisfies the conditions mentioned in Section 3.1.1.

When we have introduced multiple refinement levels, we have assured that the algorithm correctly follows the packets across all the levels. In PCRASH-AMR we have the additional complication that boxes at different levels might lie in different cores, and thus, during one time step, the ray can traverse multiple levels lying across multiple cores. Having to communicate a single ray every time it crosses domains can severely impact the code. Not only do we need frequent communication among cores, which is in itself very inefficient, but the size of the data required to communicate a single packet is also very small. Given that there is a fixed overhead of setting up a communication among cores, this will result in a scenario where this overhead is larger than communicating the data itself. In order to get around this problem, we have adopted a new scheme, similar to that in [144]. According to this, at each dt, we emit not one but multiple photon-packets, N_P, from each source. To set the value of N_P, the value of N_t is set explicitly as a user-defined value while setting up the ICs. This is different to the scheme in CRASH-AMR where N_t was set implicitly, (Section 3.1.1 has more details). The value of N_P is now given by $N_p \cdot N_s/N_t$. With this new scheme, we emit and consequently propagate a larger number of photon-packets per time step. This reduces the number of communications required per t_j. N_t is typically set to 10^5 for a simulation with $N_p = 10^8$, then 1000 packets are emitted from each source per t_j.

4.2.3.3 Improvement to the ray-tracing routine in PCRASH-AMR

In CRASH-AMR, the ray-tracing algorithm has to check for a neighbor, child or parent for each cell that is traversed. We mentioned some of the criteria that have to be taken care of in Section 3.2.2. Since in PCRASH-AMR the boxes are distributed across cores, the ray propagation is much more complex. The ray can cross multiple domains in the same time step which requires multiple communication events per dt. To resolve this problem, we have tried to make some improvements to the ray-tracing algorithm by exploiting the nesting criteria mentioned in Section 2.2.1.2.1. This restricts the ray propagation to the same level or to a finer/coarser level.

As in the serial case, we follow a ray till

(a) It reaches the end of a box.

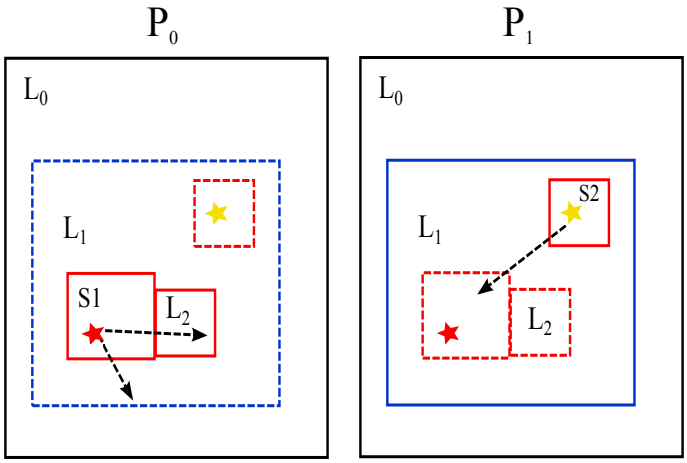

Figure 4.4 Illustration to show ray propagation across different refinement levels, distributed across cores P_0 and P_1. The different levels $L0, L1$ and $L2$ are indicated by black, blue and red boxes respectively. Local boxes are shown with solid lines and non-local boxes in dotted lines. Two sources $S1$ and $S2$ emit rays that travel through multiple levels.

(b) It enters/exits a particular refinement level.

(c) It is completely absorbed - this requires no further discussions.

Cases (a) and (b) need further checks which we discuss below. We also show a flowchart in Figure 4.5 that outlines the steps required. We refer to the flowchart as and when required.

(1) **Neighbor search:** If the ray exits a box, use the cell coordinates to check if it has entered any of the local neighbors. If so, we move the ray to the local neighbor and check if the new cell is refined or not. If so, we need to move the ray to a finer level. Here two scenarios can apply, i.e. the box containing the child cell with the ray is local or non-local.

   (i) If the child is non-local, we add the photon-packet corresponding to the ray to a linked list for now.

   (ii) If the child is local, we move the ray to a finer level and continue propagation as before.

   The steps above are shown in red boxes in Figure 4.5. The violet colored boxes show the conditions checked to determine a local or non-local child.

(2) **Parent search:** If the box has no local neighbors, we check the level below (i.e. less refined) for any local parent. If found, the coordinates of the ray are scaled to the parent level, and we again check if the cell where the ray now lies is refined. If so, we need to move the ray to a finer level. Here again two scenarios can apply, i.e. the box containing the child cell with the ray is local or non-local.

   (i) If the child is non-local, we need not look in the local map any further, since we already looked at this level in step (1). We add the photon-packet corresponding to the ray to a linked list.

(ii) If the child is local, we move the ray to a finer level and continue propagation as before.

The green boxes in Figure 4.5 show the steps taken.

(3) **Local map:** If the new box is not found in the neighbors or parents, then we search the local map. The "proper-nesting" criteria of the boxes, Section 2.2.1.2.1 has more details, allows us to improve this search. We keep a track of the refinement levels pertaining to the boxes checked in steps 1 and 2. Then it is enough if the local map is searched only among these levels, as the ray can either move to a child, neighbor or a parent level. This reduces the need to look through the whole local map to find the right box. If the box is not found within these levels, it can be assumed that the right box must be in the global map.

The blue boxes in Figure 4.5 show the steps taken.

Figure 4.4 now shows ray tracing across multiple levels, with the boxes distributed across cores P_0 and P_1. Boxes at level $L0, L1$ and $L2$ are shown as black, blue and red boxes respectively. The local boxes are shown in bold lines, non-local boxes are shown with dotted lines. One of the rays emitted from source $S1$ travels to a neighboring box which is local and so the ray can be moved to it. The second ray travels to a box at level $L1$ which lies in P_1, we search the global map only at level $L1$ to find the right box. Similarly for the ray emitted by source $S2$, it travels to a box at $L1$ which is local and then again to a box at $L2$. The box at $L1$ does not have any neighbors, so we look in the global map at level $L2$ to find the right box and communicate the packet.

(4) **Global map:** If the ray is not found in the local map, we add the photon-packet to a linked list for now and search it in the global map later. The orange boxes in Figure 4.5 show the relevant checks.

(5) At the end of a loop over all photon-packets, when all the packets have been propagated till the end of the core domain, we traverse the linked list containing the packets that need to be communicated. Each node in the linked list has the following information

(i) The new cell coordinates that the ray goes to next.

(ii) The refinement level the ray was in when it exited the box.

(iii) The data structure `PHOTON_PACKET` as mentioned in Section 3.1.2.

We do not need the source information as once the packet has been emitted in a random direction along a ray, its direction is unchanged.

(6) We now search through the global map, for all the packets in the list, to find the right core to communicate the packets to. Here as well, we search the map only among the levels necessary. If the packet was last traversing through a refinement level, say L_i, then we need to search only the levels L_{i-1}, L_i and L_{i+1} to find the right box. Once found, we know the core it belongs to and add the packet to a linked list corresponding to this core. Once all the packets have been put in the right lists, the data in the linked list is packed and communicated.

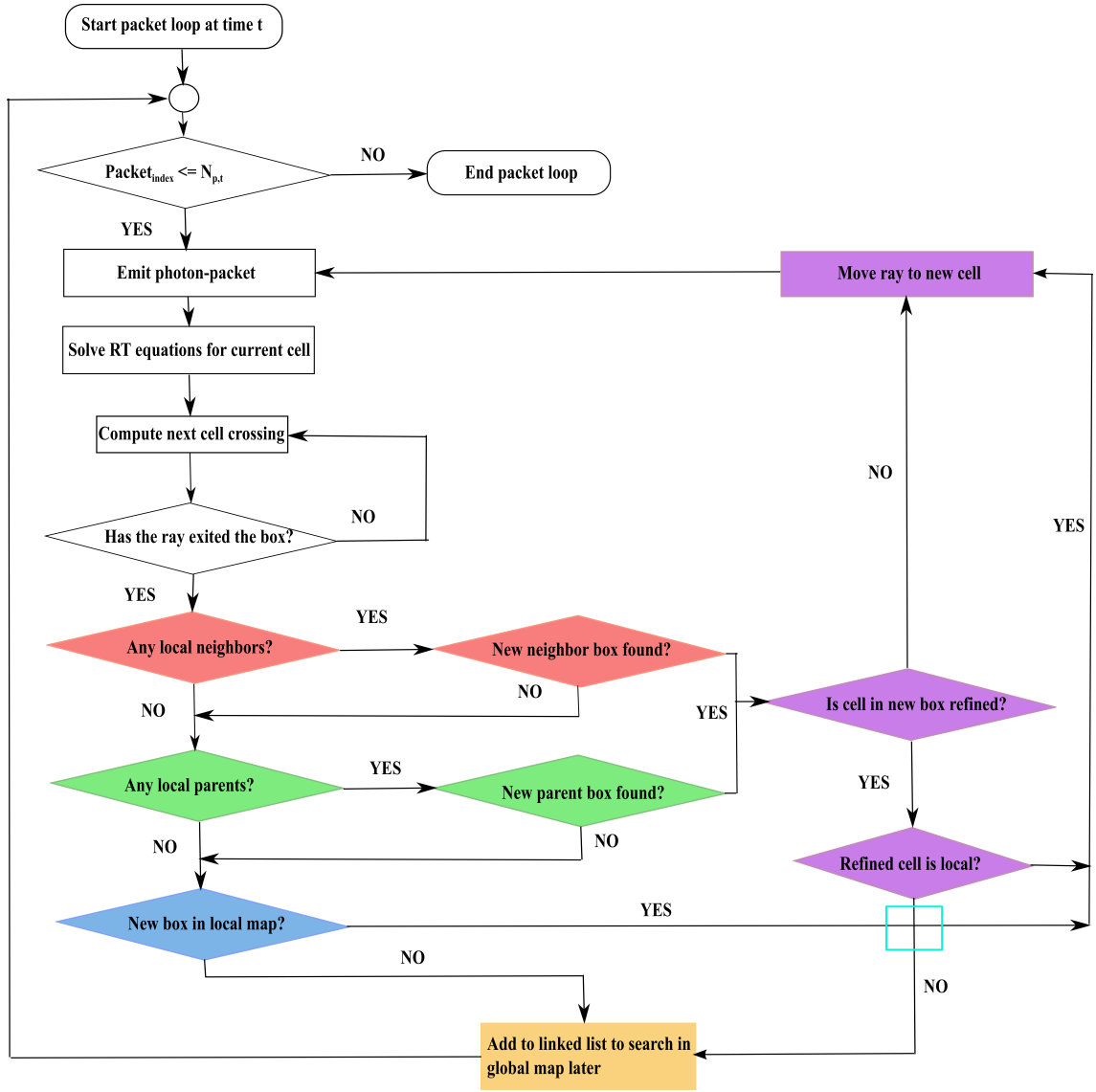

Figure 4.5 Flowchart to show the various steps now taken in the ray-tracing algorithm over multiple cores, as mentioned in Section 4.2.3.3. The boxes are colored according to the steps followed for a neighbor search (red), parent search (green), local map search (blue), checks common to all the three steps (violet) and linked list for global map (orange).

Communicating linked lists can be done in two ways. The first method is to create an MPI datatype, for example the `MPI_Type_create_struct`, which requires determining the displacement in addresses of the nodes in the linked list and committing the datatype. This can be efficient only when the size of data being sent remains constant, which does not apply to our case. In fact, we might have a different number of photon-packets that require communicating in each time step, which implies that the number of nodes in the linked list is also not constant. The second approach, which we have adopted, is to pack the list data into arrays and communicate them. In order to prevent frequent allocation and deallocation of the arrays being communicated, we follow a simple heuristic of allocating enough space for communicating 100 packets. If the number of packets being communicated, N_{comm}, becomes larger than 100, we deallocate the arrays and allocate enough space for data in $2N_{comm}$ packets and so on. Since the number of nodes being sent/received is known, the send/receive arrays can be indexed into appropriately to get the right data.

Once the packets have been received, they are propagated further till they reach the end of the core domain. If so, then we follow the steps outlined earlier and repeat this till all the photon-packets emitted in one time step have been propagated. At the end of each dt, we check if t_j coincides with the time at which an output file has to be written out using the I/O routines in `CHOMBO`. If not then we proceed to the next time step. If so then the output files are written out, we next discuss the changes done in the I/O routines for `PCRASH-AMR`.

4.2.4 `CHOMBO` I/O

We mentioned in Section 3.2.3 that `CHOMBO` uses the HDF5 library to write output files. When in parallel mode, `CHOMBO` makes use of the collective mode, i.e. *H5FD_MPIO_COLLECTIVE*, as the "data transfer property list" to write files. In this case the local boxes at each level are accessed by each core, using the *DataIterator* class, and written out. This works well as long as all the cores contain an equal number of boxes at each refinement level. However, there is no guarantee that it would hold true for all cases, and in fact when using the criteria of ray density as load balancing, we encountered a situation where the cores did not contain an equal number of boxes, resulting in a deadlock. We then changed the I/O routines to use the *H5FD_MPIO_INDEPENDENT* mode instead, which resolved the issue.

This completes our discussion of the changes done to the ray-tracing routine and the other `CHOMBO` related changes in `PCRASH-AMR`. We next look at the changes done with respect to RNGs which are important for our MCRT simulation.

4.2.5 Parallel RNGs

The main algorithm in `PCRASH-AMR` works on the principles of MC sampling. For a parallel MC code, it is important to employ good quality PRNGs that can generate independent streams of RNs. `CRASH-AMR` uses a simple PRNG, *ran2* for generating rays in random directions. This is not sufficient when the code is parallelised, as we need to generate independent random numbers (RN) among the cores. We have enabled `PCRASH-AMR` to use two parallel PRNGs (PPRNGs), the Scalable Parallel RNG library (`SPRNG`) [121] and the `RngStream` library by L'Ecuyer [108], due to their availability as a complete PPRNG library.

SPRNG provides six different PPRNGs that the user can specify while initialising the library. The library has been found to pass a number of empirical tests [121] and provides good quality RNs. The RngStream library supports multiple independent streams of RNs and has also been proven to pass many statistical tests suggested by [100] and [120].

To interface these libraries with PCRASH-AMR we have modified the SYS_RANDOM module of the code to accept the two libraries as an option. Using SPRNG in a parallel mode is simple and only requires defining the *USE_MPI* variable along with the same seed being used to initialise the PPRNG. RNGStream, on the other hand, needs some slight modifications to be able to generate parallel streams.

The RngStream library creates a new stream on each core from the call to the function *RngStream_CreateStream()*. The library does not encourage the use of the same seed on different cores and hence we had to adopt a different strategy to generate new seeds. [94] suggest to use a subroutine within the RngStream library itself to advance the seeds to the next state. The library has a subroutine *MatVecModM()* which advances the seed array to the next state after creating a stream and is useful if multiple streams are created by the same process. It also ensures independent streams of RNs among cores. We have used this subroutine to get new seed values to ensure that independent streams are generated on multiple cores.

Before concluding this Section we briefly describe the software architecture of the code. The development of PCRASH-AMR also follows the methodologies used in developing CRASH-AMR. We provide the user with options to enable or disable the parallel part of the code during compile and run time. The changes to the SYS_RANDOM module have already been extensively discussed in Section subsection:PRNG. Similar to the AMR_PARAM datatype, we define PAMR_PARAM for the parallel modules that contains information regarding the MPI communicator, the load balancing criteria to be used and the number of packets being sent/received every time step. This completes the description of the new code PCRASH-AMR. We have tested this code with different test scenarios to study its performance, these are discussed in the next Section.

4.3 Test scenarios

In this section we show the results of some of the tests we have done to study the performance of the code on two different machines, *Odin* and *Hydra* at RZG, Garching. *Odin* has 156 execution hosts with each node containing two Intel E5-2670 processors (8 cores per processor) with 2.6 GHz CPUs. *Hydra* on the other hand has two parts, the first is the Sandy Bridge part containing 628 compute nodes with two Intel ES-2670 processors per node. The second is the Ivy Bridge part that has 3690 compute nodes with 20 cores in each node operating at 2.8 GHz. Our tests done on Hydra use the Ivy Bridge part of the machine where ever the number of cores being used is 64 or more. *Odin* provides Intel MPI whereas on *Hydra* IBM MPI is available by default.

We have set-up test cases similar to the ones discussed in Section 3.3. We begin with a standard test case to verify the correctness of the code when multiple cores are being used. Then we perform test cases with realistic density fields, and we look at the performance of

the code when AMR grids with 3 and 4 refinement levels (r.l.) are used. Note that for all test cases, the following criteria apply

1. The scaling tests have been run on 1, 16, 32, 64 and 128 cores, unless specified.
2. The results with the RD calculation disabled and enabled are presented for 16 cores and above, as this is not applicable for a serial case. When RD is disabled or switched off, the default LB routine of CHOMBO is used.
3. N_p is set to 10^8 and t is set to 500 Myr for all cases, unless specified.
4. To generate RNs the Multiplicative Lagged Fibonacci Generator (MLFG) of the SPRNG library has been used. We ran some sample tests to study the efficiency of the different PPRNGs provided by the two libraries mentioned in Section 4.2.5, by generating 10^{12} RNs and comparing the run times. The MLFG was found to be the most efficient, along with the Combined Multiple Recursive Generator (CMRG) of SPRNG.
5. N_t has been set to 10^5 in all cases. N_t is inversely proportional to the time resolution of the simulation, mentioned in Section 3.1.1, and also decides the value of N_P. The value of N_t chosen is high enough to ensure that the time resolution remains fine enough for the RT equations to be solved correctly. The value is also low enough to ensure that a sufficient number of photon-packets is communicated at each time step, which is 1000 in our case.

Before we discuss the tests, we mention the naming convention followed for referring to the tests in the rest of this Section. The tests are named with the following abbreviations

1. O (Odin) or H (Hydra)
2. 3 or 4 to denote the number of refinement levels
3. S (serial) or P (Parallel), to refer to CRASH-AMR or PCRASH-AMR, followed by the number of PEs the test was run on.

For example, *O3S1* refers to a test run on *Odin* for 3 r.l. with CRASH-AMR (1 core). *H4P1* refers to a test run on *Hydra* for 4 r.l. with PCRASH-AMR (1 core).

Additionally, we use W or S to refer to weak or strong scaling tests on multiple cores. For the weak scaling case, the problem size is increased together with the number of cores. For strong scaling, the problem size is kept fixed on an increasing number of cores. For example, *WO3P16* refers to a weak scaling test case, run on *Odin* with 16 PEs. Similarly, *SH4P128* refers to a strong scaling test case run on *Hydra* with 128 PEs.

4.3.1 Test 1: Strömgren sphere in a H medium

We have set up a test equivalent to Test 1 of the RTCCP, as discussed in Section 3.3.1. The test simulates the evolution of an ionised region around a single point source located at the grid origin (0,0,0) in a cosmological box of side length $L_{box} = 6.6$ kpc (comoving). In order to

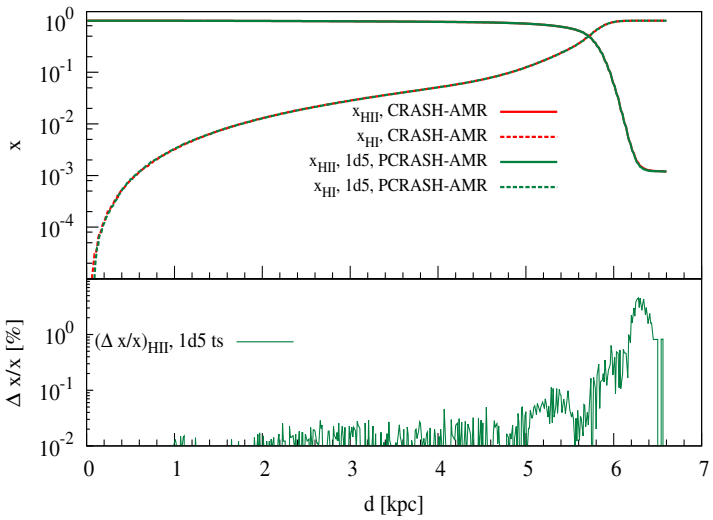

Figure 4.6 Spherically-averaged profile at time $t = 500$ Myr, at d [kpc], for Test 1. The colors refer to `CRASH-AMR` (red) and `PCRASH-AMR` (dark-green). **Top:** Profiles of $x_{\rm HII}$ (solid lines) and $x_{\rm HI}$ (dashed lines). **Bottom:** Δ between the `CRASH-AMR` and `PCRASH-AMR` results.

be able to distribute the boxes among cores, this test case has been set-up on a uniform grid of resolution 512^3 cells. The domain has been split into 512 equal sized boxes with `CHOMBO` routines, and the LB routines mentioned in Section 4.2.1 have been used to distribute the load on multiple cores. This test has been run only on *Odin* for verifying the correctness of the code.

To begin with, we compare the results of running `PCRASH-AMR` on one core to those from `CRASH-AMR`. Note that in `CRASH-AMR` the RNs are generated with the *ran2* function, while in `PCRASH-AMR` the MLFG provides the RNs. This decides the random directions that the rays travel in, and consequently the cells they ionise. Also, the time resolution for the serial and parallel test case is different, in the former case we emit one photon-packet per dt. In the latter case, we emit 1000 photon-packets per dt, hence the cell crossing times in both scenarios are different. Since the test has an analytical solution we expect some, although minimal, difference in the results. The outcome is shown in Figure 4.6, where the panel shows the spherically-averaged physical quantities as a function of d [kpc], together with the Δ values calculated using R_{ref} for results of `CRASH-AMR`, and R_i for results of `PCRASH-AMR`.

We find that the extent of the $x_{\rm HII}$ region is the same for both codes, i.e. $\sim$5.4 kpc, as per the analytical solution. In the partially ionised regions, beyond 5.4 kpc, Δ is as high as 5 % for $x_{\rm HII}$. As mentioned earlier, this is due to the different RNGs that have been used and the time resolution for each test case.

The next step was to run the same set-up on multiple cores. Since, the test case requires only one point source, the cores that are assigned boxes further away from the source will not have any packets to process till the I-front reaches that portion of the domain. As a result, during early times in the RT simulation, there will be some idle cores. Also, only one core contains the box with the point source, and so the PPRNG does not have to generate independent

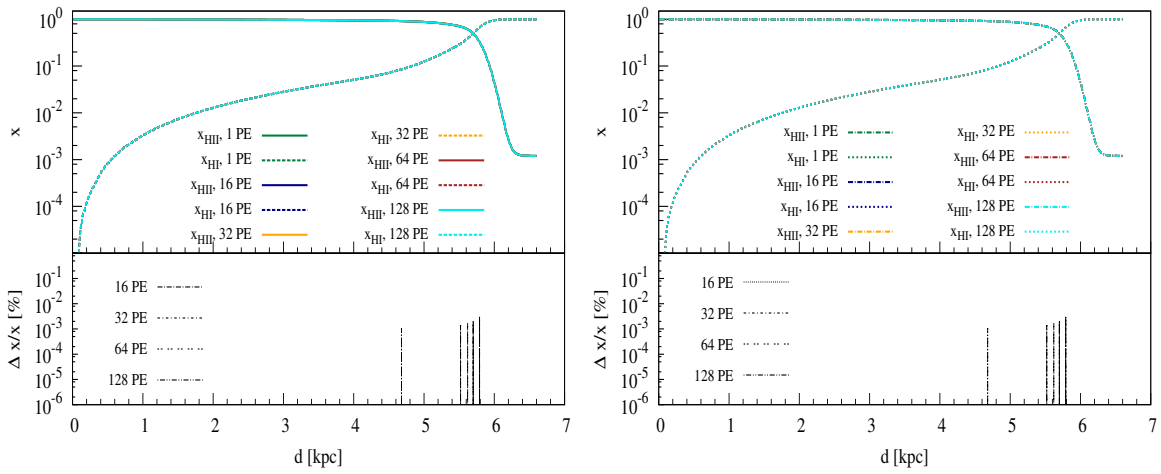

Figure 4.7 Spherically-averaged profile at time $t = 500$ Myr, at d [kpc], for Test 1. The colors refer to PCRASH-AMR run on 1 PE (dark-green), 16 PEs (blue), 32 PEs (orange), 64 PEs (brown) and 128 PEs (cyan). **Left:** Profiles of $x_{\rm HII}$ (solid lines) and $x_{\rm HI}$ (dashed lines) with RD disabled. **Right:** Profiles of $x_{\rm HII}$ (dash-dot lines) and $x_{\rm HI}$ (dotted lines) with RD enabled. **Bottom sub-panels:** Δ between the PCRASH-AMR results.

streams of RNs. This essentially means that the cells crossed by the rays traveling in random directions will be the same in all test cases and we don't expect to see any differences in the volume averaged $x_{\rm HII}$ fractions at the end of the RT simulation.

Figure 4.7 shows the spherically-averaged physical quantities as a function of d [kpc], together with the Δ values. We show the results for cases when RD is disabled and enabled. The Δ values in both cases are calculated using R_{ref} for results of PCRASH-AMR run on one core or processing element (PE), R_i refers to the results of PCRASH-AMR run on multiple cores.

We find that the results agree well among all the cores, for both settings of RD, with a maximum Δ value of only 0.001. Figure 4.8 shows the speed-up obtained for Test 1 when RD is disabled (blue) and enabled (cyan). The speed-up has been calculated using the total elapsed time till the end of the simulation at $t = 500$ Myr, with the serial case as the baseline. As mentioned earlier, this test case has only a single point source, so the load imbalance between the cores is very pronounced when RD is disabled. The maximum speed-up obtained is only 1.72 on 128 cores. Even with RD enabled, the maximum speed-up obtained is 2.36 on 16 cores and increases to 2.43 at 128 cores.

4.3.2 Test 2: Realistic density field with one point source

The set-up for this test is similar to the test in Section 3.5.1 with the ionised region from a single point source expanding into a realistic density field. The simulation was run on *Odin* and *Hydra* with the ICs set on a grid with 3 r.l., and N_p was set to $2 \cdot 10^8$ packets. Similar to Test 1, we have only one point source in the whole domain. So only one core will emit the photon-packets while the rest will be idle till the I-front reaches their domain. Also, we have the additional complexity of propagating the ray through multiple levels. Overall, the

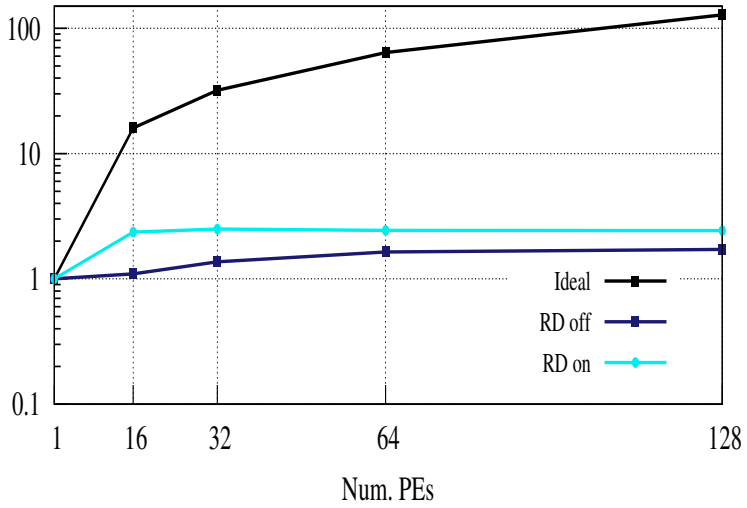

Figure 4.8 Speed-up obtained by running `PCRASH-AMR` for Test 1 on different number of PEs on *Odin*. The colors refer to RD disabled (blue) and RD enabled (cyan).

test case is poorly load balanced due to the set-up used. Nevertheless, we need to verify that with multiple r.l., and multiple cores, `PCRASH-AMR` is able to provide results consistent with those of `CRASH-AMR`.

We begin by comparing the results obtained in Section 3.5.1 with the results from `PCRASH-AMR`. Table 4.1 shows the volume averaged $x_{\rm HII}$ values at time $t = 200,\ 500$ Myr when RD is disabled and enabled, for *Odin*. Table 4.2 shows the corresponding results for *Hydra*. We show the results, on both machines, for `CRASH-AMR` (O3S1, H3S1) and `PCRASH-AMR` run on 1 (O3P1, H3P1), 16 (O3P16, H3P16), 32 (O3P32, H3P32), 64 (O3P64, H3P64) and 128 (O3P128, H3P128) cores.

We find that the results agree well between all the test cases with only a maximum difference of 0.14% and 0.05% for the volume averages at $t = 200$ and 500 Myr respectively. We next look at the speed-up obtained with this test case. Figure 4.9 shows the speedup obtained by running the test on multiple cores on *Odin* and *Hydra*. The speed-up has been calculated similar to Section 4.3.1.

On *Odin*, when RD is disabled, the maximum speed-up is only 1.33 at 128 cores. Although the boxes are distributed according to their loads, the set-up results in the cores containing the boxes near the source doing most of the work while the others remaining idle. Hence we do not see much of a speed-up with increasing number of cores. When RD is enabled, the maximum speed-up is 1.34 at 32 cores and decreases to 1.26 at 128 cores. When the LB is done using RD, the boxes near to the source have a higher load and are distributed among different cores rather than being assigned to the same core. The rays, as a result, have to be communicated to these cores for further propagation. However, intra-node MPI communication is not expensive and so we see some speed-up on 16 cores, a factor of 1.32, when compared to running on one core. With increasing number of cores, inter-node MPI communication dominates and the cores spend a considerable amount of time in communication.

Table 4.1. Volume averaged $x_{\rm HII}$ values for Test 2, with ICs set on grids with 3 r.l., on *Odin*. The results are from CRASH-AMR and PCRASH-AMR simulations at time $t = 200,\ 500$ Myr. We compare the results between test cases that are at the same t.

Test case	$x_{\rm HII}$ ($t = 200$ Myr)		$x_{\rm HII}$ ($t = 500$ Myr)	
	RD off	RD on	RD off	RD on
O3S1	0.0677	0.0677	0.1806	0.1806
O3P1	0.0677	0.0677	0.1806	0.1806
O3P16	0.0676	0.0677	0.1805	0.1805
O3P32	0.0677	0.0677	0.1805	0.1806
O3P64	0.0677	0.0677	0.1806	0.1806
O3P128	0.0677	0.0677	0.1805	0.1806

Table 4.2. Volume averaged $x_{\rm HII}$ values for Test 2, with ICs set on grids with 3 r.l., on *Hydra*.

Test case	$x_{\rm HII}$ ($t = 200$ Myr)		$x_{\rm HII}$ ($t = 500$ Myr)	
	RD off	RD on	RD off	RD on
H3S1	0.0677	0.0677	0.1806	0.1806
H3P1	0.0677	0.0677	0.1806	0.1806
H3P16	0.0676	0.0677	0.1805	0.1805
H3P32	0.0677	0.0677	0.1805	0.1806
H3P64	0.0677	0.0677	0.1806	0.1806
H3P128	0.0677	0.0677	0.1805	0.1806

On *Hydra*, the speed-up on 128 cores is 1.71 with RD is disabled, and 1.66 with RD enabled. The same argument, as for *Odin*, applies for the difference in speed-up factors obtained for the two LB methods. Between the machines themselves, we find that *Hydra* has a 7% higher run time than *Odin* on 128 cores. This can be attributed to the node configuration on the Ivy Bridge part of Hydra which has 20 cores per node instead of 16 as on Odin. We need to assign 8 cores to the first node before filling up the rest of the allocated nodes completely. The additional inter-node communication required to propagate the rays contributes to the higher run times.

4.3.3 Test 3: Realistic density field with multiple point sources

Our next tests have been to run PCRASH-AMR on multiple cores, with multiple point sources. We use the outputs from the CHARM simulations and run the RT simulations on grids with

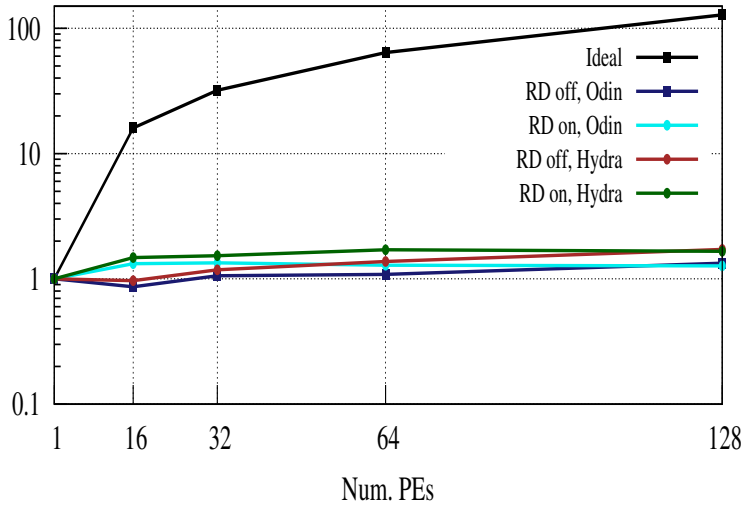

Figure 4.9 Speed-up obtained by running PCRASH-AMR for Test 2, on grids with 3 r.l., on different number of PEs on *Odin* and *Hydra*. The colors refer to PCRASH-AMR run on *Odin* with RD disabled (blue) and RD enabled (cyan), on *Hydra* with RD disabled (brown) and RD enabled (green).

3 and 4 r.l. The set-up of these test cases is similar to that in Section 3.3.3.1, with the difference that the luminosity of the sources, $\dot{N}_\gamma$, is set to values between $8 \cdot 10^{53}$ and $1.1 \cdot 10^{54}$ photons s^{-1}. Ideally, the luminosity of the source is proportional to $n_{\rm gas}$ values at the source location. Since the refinement criterion in CHARM is based on a density threshold we can, for the purposes of our test case, assume that the $n_{\rm gas}$ values are similar across all the boxes at the highest refinement level. Hence the luminosities have been set to fairly equal values for all sources.

We ran both weak and strong scaling tests for this set-up. In the case of weak scaling we fix the number of point sources to be equivalent to the number of cores. For the strong scaling tests, we fix the number of point sources and increase the number of cores. For the strong scaling tests, with increasing number of cores, the distribution of boxes among the cores cannot be decided a-priori. As a result, the point source locations cannot be selected such that each core has at least one source for all cases. What we do ensure, for all weak and strong scaling test cases, is that each box contains only one point source. Also the strong scaling tests have been run on a maximum of 128 cores with 128 point-sources. So for the test case with 128 cores, and RD disabled, each core has at least one point source assigned to it.

4.3.3.1 Test 3a: Weak scaling results

The weak scaling results for both *Odin* and *Hydra* are discussed in this section. The tests have been run on 16, 32, 64 and 128 cores, till time $t = 500$ Myr, with intermediate outputs at 10, 50, 100 and 200 Myr. The weak scaling efficiency has been calculated using the total

elapsed time till the end of the simulation at $t = 500$ Myr, with 16 cores as the baseline. It is necessary to mention the following. When RD is disabled, the point sources can be set at locations such that each core has at least one point source. When RD is enabled though, the boxes get reassigned among the cores as per the load calculated using the scheme in Section 4.2.1. Hence, in this case it is not possible to ensure that each core contains at least one source without changing the point source locations again. Due to the varying distribution of sources between the two LB methods, the core can be allocated with one or multiple sources. Consequently the RNs generated by the PPRNG will be different. Therefore, it is essential that the results of using both LB methods are compared to ensure that PCRASH-AMR provides consistent results for different set-ups run on the same number of cores.

Before we discuss the results, two important points need to be noted here

1. The number of point sources for each test case depends on the number of cores. So, we compare the results only between relevant test cases. For example, *WO3P16* has 16 sources and should not be compared with *WO3P128* which has been set-up with 128 sources. The volume averaged fractions will obviously be different.

2. The location of the point sources for test cases at different refinement levels do not have any correlation. This is different to the set-up in Section 3.3 where the point source locations had been set such that when moving from higher to lower refinement levels, the luminosities were added if two sources fell in the same box. The scenario for our parallel tests is different, we would like to ensure that each box has only one point source, in order to distribute the load accordingly. This can be properly ensured only if we explicitly set the sources for different refinement levels. Hence, we do not compare the volume averaged results between tests done with different refinement levels. For example, we do not compare *WH3P16* with *WH4P16* even though both test cases have 16 sources.

Table 4.3 shows the volume averaged x_{HII} values at time $t = 200,\ 500$ Myr when RD is disabled and enabled, for tests with 3 r.l. on both machines. The results on *Odin* for 16 (*WO3P16*), 32 (*WO3P32*), 64 (*WO3P64*) and 128 (*WO3P128*) cores are shown. The corresponding results on *Hydra* for 16 (*WH3P16*), 32 (*WH3P32*), 64 (*WH3P64*) and 128 (*WH3P128*) cores are also shown. We compare the volume averaged values obtained for the same number of cores, for the two LB methods. For example, the results of (*WO3P16*) at $t = 200$ Myr for both LB methods are compared. Overall, we find a difference of less than 0.01% between the test cases run on the same machine. Between *Odin* and *Hydra* as well the difference in results between the corresponding test cases, for example *WO3P16* and *WH3P16*, is less than 0.01%.

Table 4.4 shows the corresponding results with 4 r.l. The results on *Odin* for 16 (*WO4P16*), 32 (*WO4P32*), 64 (*WO4P64*) and 128 (*WO4P128*) cores are shown. The results on *Hydra* for 16 (*WH4P16*), 32 (*WH4P32*), 64 (*WH4P64*) and 128 (*WH4P128*) cores are also shown. The maximum difference in values between the test cases run on the same machine is 0.01%. Between *Odin* and *Hydra*, the difference in results is about 0.1%. We then conclude that PCRASH-AMR is able to provide consistent results for all the test cases run on different machines.

We now discuss the weak scaling results. Figure 4.10 shows the results for tests run with 3 and 4 r.l., till time $t = 500$ Myr for both *Odin* and *Hydra* when RD is disabled and enabled.

Table 4.3. Volume averaged $x_{\rm HII}$ values for Test 3a, with ICs set on grids with 3 r.l, on *Odin* and *Hydra*. The results are from `PCRASH-AMR` simulations at time $t = 200,\ 500$ Myr. The number of point sources increases with number of cores. We compare results between test cases at same t, with RD off and on, for the same number of cores.

Test case	$x_{\rm HII}$ ($t = 200$ Myr)		$x_{\rm HII}$ ($t = 500$ Myr)	
	RD off	RD on	RD off	RD on
WO3P16	0.4232	0.4231	0.7265	0.7264
WO3P32	0.5646	0.5646	0.8209	0.8208
WO3P64	0.7427	0.7427	0.9214	0.9214
WO3P128	0.8984	0.8984	0.9774	0.9774
WH3P16	0.4232	0.4231	0.7265	0.7264
WH3P32	0.5646	0.5646	0.8209	0.8208
WH3P64	0.7427	0.7427	0.9214	0.9214
WH3P128	0.8984	0.8984	0.9774	0.9774

Table 4.4. Volume averaged $x_{\rm HII}$ values for Test 3a, with ICs set on grids with 4 r.l, on *Odin* and *Hydra*. The results are from `PCRASH-AMR` simulations at time $t = 200,\ 500$ Myr. The number of point sources increases with number of cores. We compare results between test cases at same t, with RD off and on, for the same number of cores.

Test case	$x_{\rm HII}$ ($t = 200$ Myr)		$x_{\rm HII}$ ($t = 500$ Myr)	
	RD off	RD on	RD off	RD on
WO4P16	0.1935	0.1935	0.5353	0.5353
WO4P32	0.4511	0.4512	0.7813	0.7813
WO4P64	0.7062	0.7062	0.9115	0.9115
WO4P128	0.8881	0.8881	0.9735	0.9735
WH4P16	0.1935	0.1934	0.5353	0.5353
WH4P32	0.4511	0.4511	0.7812	0.7812
WH4P64	0.7061	0.7061	0.9115	0.9115
WH4P128	0.8868	0.8868	0.9732	0.9732

Table 4.5. Weak scaling run times (in minutes) for Test 3a, with ICs set on grids with 3 and 4 r.l., on *Odin* and *Hydra*. The timings are for the simulations run till $t = 500$ Myr.

Test case	3 r.l.		Test case	4 r.l.	
	RD off	RD on		RD off	RD on
WO3P16	420.76	383.22	WO4P16	345	419.4
WO3P32	408.76	542.3	WO4P32	517.4	660.25
WO3P64	542	735	WO4P64	756	840.28
WO3P128	790	915	WO4P128	960.2	1066
WH3P16	523	471	WH4P16	424.51	511.51
WH3P32	506	652	WH4P32	634.5	811.2
WH3P64	592	805	WH4P64	828	913.35
WH3P128	849	1062	WH4P128	1009.83	1164

For completeness, we also provide the total run times (in minutes) for all test cases in Table 4.5.

We first discuss the results for test cases run on grids with 3 r.l. When RD is disabled, on *Odin*, the weak scaling efficiency is 77.6% for *WO3P64* and drops to 53.2% for *WO3P128*. The corresponding values on *Hydra* are 88.3% for *WH3P64* and 61.6% for *WH3P128*. Even though the load in this case is equal on all cores, photon-packets from multiple sources need to be communicated multiple times per dt. This begins to affect the efficiency as we go to a larger number of point-sources. When RD is enabled, the efficiency is 52.1% for *WO3P64* and 41.9% for *WO3P128*. *Hydra* shows an efficiency of 58.5% for *WH3P64* and 44.3% for *WH3P128*. The efficiency when RD is disabled, is higher by a factor of 1.5 and 1.3 for 64 and 128 cores respectively, on both *Odin* and *Hydra*, compared to when RD is enabled. This difference arises due to the fact that with RD enabled, the boxes near the source are distributed among the cores requiring more communication to propagate the rays. Also, as mentioned earlier, using RD does not ensure that each core has at least and at most one point-source, hence the load is not distributed equally. Finally note that the total run time on *Hydra* for 16 cores, where the Sandy Bridge part of the machine is used, is higher than *Odin* by a factor of ~ 1.25. At 128 cores, this run time is higher by a factor of 1.1. Note that the MPI environment is different in the two machines, even though the same processors are used in the test case with 16 cores. Additionally, the difference for 128 cores can be explained using the same argument as in Section 4.3.2, i.e. the difference in node configuration.

Next, we look at the results for test cases run on grids with 4 r.l. When RD is disabled, the weak scaling efficiency on *Odin* is 45.6% on 64 cores and drops down to 36% on 128 cores. The corresponding values on *Hydra* are 51.2% for *WH3P64* and 42% for *WH3P128*. With RD enabled, the efficiency is 50% and 39.3% for 64 and 128 cores, respectively on *Odin*. The corresponding values on *Hydra* are 56% and 44%. Although the scaling efficiency for all test cases with RD on seems to be higher, in terms of run times it is a factor of ~ 1.2 higher than the test cases where RD is disabled. Here as well, we find that the run time on *Hydra* for 16

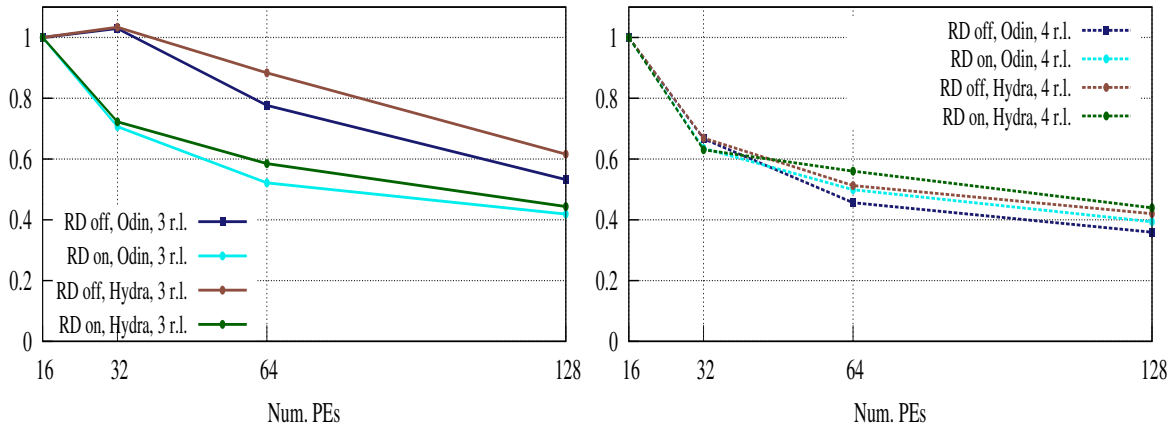

Figure 4.10 Weak scaling efficiency for Test 3a, with ICs set on grids with 3 and 4 r.l., on *Odin* and *Hydra*. The efficiency is shown for simulations run till $t = 500$ Myr. The colors refer to `PCRASH-AMR` run on *Odin* with RD disabled (blue) and RD enabled (cyan), on *Hydra* with RD disabled (brown) and RD enabled (green). **Left:** Efficiency for tests run on grids with 3 r.l (solid lines). **Right:** Efficiency for tests run on grids with 4 r.l (dotted lines).

cores is higher than *Odin* by a factor of ~ 1.25. At 128 cores, this run time is higher by a factor of 1.1.

We now discuss the efficiency of the code when the number of refinement levels are increased. As mentioned earlier, this is not an ideal comparison, as the point source locations for the test cases with 3 r.l. do not correspond to those with 4 r.l. since the placement is arbitrary. We do not ensure that a source placed in a box at the 3rd r.l. is placed at the same location in a box at the 4th r.l. What we can do, in general, is to look at the trend in these test cases. From Table 4.5, for *Odin*, it is evident that the efficiency of the parallel code drops when the number of levels is increased. With RD disabled, the run times for *WO4P64* and *WO4P128* is higher by a factor of 1.5 and 1.2 when compared to *WO3P64* and *WO3P128*. When RD is enabled, this factor is 1.2. The trend is the same for *Hydra* as well. The reason for this drop is due to the fact that the rays now need to propagate through an additional level, where again the neighboring, child or parent boxes can be local or non-local. This will again involve communication among cores to propagate the ray further.

We next look at the results from our strong scaling tests.

4.3.3.2 Test 3b: Strong scaling results

We discuss the strong scaling results for both *Odin* and *Hydra* in this section. The tests have been run on 16, 32, 64 and 128 cores till $t = 100$ Myr on both machines. The speed-up has been calculated using the total elapsed time till the end of the simulation at $t = 100$ Myr, with 16 cores as the baseline. All test cases have been run with 128 point sources set at the highest refinement level. We apply the same arguments as given in Section 4.3.3.1 to compare the results of the different test cases and verify the correctness of `PCRASH-AMR`.

Table 4.6. Volume averaged $x_{\rm HII}$ values for Test 3b, with ICs set on grids with 3 r.l, on *Odin* and *Hydra*. The results are from PCRASH-AMR simulations at time $t = 100$ Myr. The number of point sources is 128, and is fixed for all test cases. The placement of sources for tests with 3 r.l. do not correspond to the tests with 4 r.l. We compare the results between test cases run on grids with same number of r.l.

Test case	$x_{\rm HII}$ ($t = 100$ Myr, 3 r.l.)	
	RD off	RD on
SO3P16	0.7528	0.7528
SO3P32	0.7529	0.7529
SO3P64	0.7529	0.7529
SO3P128	0.7528	0.7529
SH3P16	0.7528	0.7528
SH3P32	0.7529	0.7529
SH3P64	0.7529	0.7529
SH3P128	0.7528	0.7529

Table 4.6 shows the volume averaged $x_{\rm HII}$ values at time $t = 100$ Myr when RD is disabled and enabled, for tests with 3 r.l, the corresponding results for tests with 4 r.l are shown in Table 4.7. We compare the volume averaged values obtained on all cores for the same number of refinement levels, for the two LB methods. For example we compare *O3PE16* and *O3PE128* with *H3PE16* and *H3PE128*. For tests with 3 r.l., we find a difference of 0.01% between all the test cases. The difference between the results with 4 r.l. is about 0.3% for all test cases run on both the machines.

Overall, we find that the results agree well between all the test cases.

We next look at the speed-up obtained from these tests, shown in Figure 4.11, run with 3 and 4 r.l. on both machines. For clarity, we show the run times in Table 4.8.

For the case with 3 r.l., we find a speed-up of 1.6, 2.36 and 3.57 on *Odin*, for 32, 64 and 128 cores, with RD disabled. The corresponding values on *Hydra* are 1.64, 2.63 and 4.1. When RD is enabled, the speed-up on *Odin* for 32, 64 and 128 cores are 1.13, 2.26 and 3.37. On *Hydra* the corresponding speed-ups are 1.3, 2.8 and 4.05. From the graphs, it seems that the test cases with RD enabled on *Hydra* perform as well as those with RD disabled. However, in terms of total run time the former is still a factor of 1.2 higher than the latter. For example, from Table 4.8, *SH3P16* has a run time of 578 (RD disabled) and 697 (RD enabled) minutes. The corresponding run times on 128 cores, *SH3P128*, are 141 and 172 minutes respectively. Also, comparing both machines we find that tests on *Hydra* have longer run times, a factor of ~ 1.2 higher, than those on *Odin*.

For the case with 4 r.l., on *Odin*, we find a speed-up of 1.63, 2.53 and 3.74 on 32, 64 and 128 cores with RD disabled. The corresponding values on *Hydra* are 1.65, 2.87 and 4.4. For RD enabled, we find a speed-up of 1.65, 2.72 and 3.92 on *Odin* for 32, 64 and 128 cores. The

Table 4.7. Volume averaged $x_{\rm HII}$ values for Test 3b, with ICs set on grids with 4 r.l, on *Odin* and *Hydra*. The results are from `PCRASH-AMR` simulations at time $t = 100$ Myr. The number of point sources is 128, and is fixed for all test cases. The placement of sources for tests with 3 r.l. do not correspond to the tests with 4 r.l. We compare the results between test cases run on grids with same number of r.l.

Test case	$x_{\rm HII}$ ($t = 100$ Myr, 4 r.l.)	
	RD off	RD on
SO4P16	0.7247	0.7246
SO4P32	0.7248	0.7247
SO4P64	0.7248	0.7248
SO4P128	0.7248	0.7248
SH4P16	0.7222	0.7218
SH4P32	0.7221	0.7220
SH4P64	0.7221	0.7221
SH4P128	0.7221	0.7222

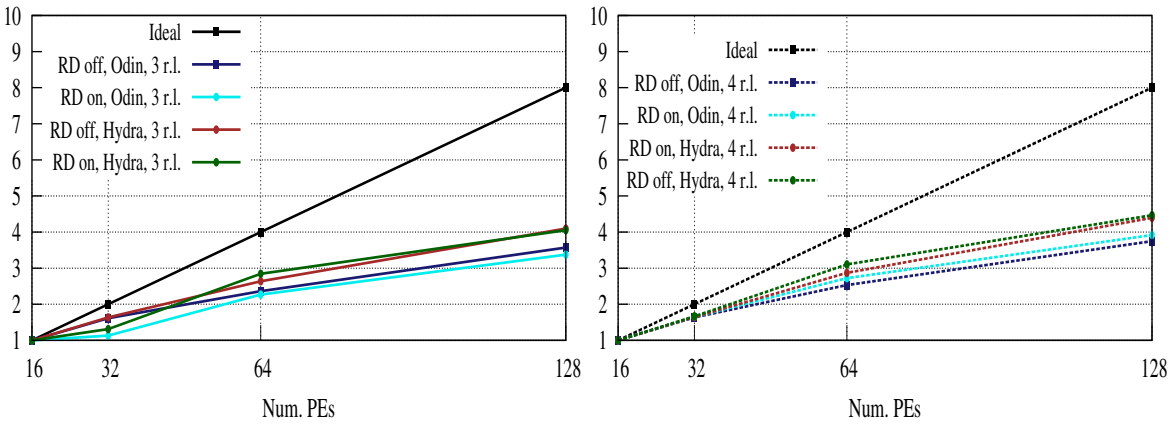

Figure 4.11 Speed-up for Test 3b, with ICs set on grids with 3 and 4 r.l., on *Odin* and *Hydra*. The speed-up is shown for simulations run till $t = 100$ Myr. The colors refer to `PCRASH-AMR` run on *Odin* with RD disabled (blue) and RD enabled (cyan), on *Hydra* with RD disabled (brown) and RD enabled (green). **Left:** Speed-up for tests run on grids with 3 r.l (solid lines). **Right:** Speed-up for tests run on grids with 4 r.l (dotted lines).

Table 4.8. Total run time (in minutes) for Test 3b, with ICs set on grids with 3 and 4 r.l., on *Odin* and *Hydra*. The timings are for simulations run till $t = 100$ Myr.

Test case	3 r.l.		Test case	4 r.l.	
	RD off	RD on		RD off	RD on
SO3P16	468	513	SO4P16	577	654
SO3P32	290	453	SO4P32	353	396
SO3P64	198	226	SO4P64	228	240
SO3P128	131	152	SO4P128	154	167
SH3P16	578	697	SH4P16	703	795
SH3P32	353	531	SH4P32	427	477
SH3P64	219	245	SH4P64	245	256
SH3P128	141	172	SH4P128	160	178

corresponding values on *Hydra* are 1.67, 3.1 and 4.47. In terms of run time, the case with RD enabled performs less efficiently than that with RD disabled, and the total run time is higher by a factor of ~ 1.1. We find the same trend in run times between the two machines, i.e. *Odin* performs better than *Hydra*.

Even though the speed-up factors for the test cases with 3 r.l. are comparable to those with 4 r.l., the total run times in the latter case are higher by a factor of 1.3, similar to our findings in Section 4.3.3.1.

4.4 Conclusion

We have discussed the development of PCRASH-AMR, a parallel version of the CRASH-AMR code. Parallelisation of a ray-tracing code in the presence of multiple refinement levels is rather complex and requires several factors to be taken into account for good performance. We have attempted to parallelise the code using distributed memory parallelism and have discussed in detail the techniques used to do so. Two factors to determine the load on the cores have been mentioned, the first being the default method in CHOMBO which uses the number of cells in a box as the load and the second method of calculating the RD in each cell as the load. We have devised methods to speed up the ray-tracing algorithm by exploiting the nesting criteria followed in CHOMBO. Two PPRNGs have also been incorporated that provide independent stream of RNs in PCRASH-AMR. The development of the code has been done following the same principles applied for CRASH-AMR.

We have tested the code using the standard test prescribed in RTCCP and compared the results with those described in Chapter 3. We have also looked at sample test cases using outputs from a realistic hydrodynamic simulation with AMR refinement.

The standard test shows a good agreement with CRASH-AMR, confirming that the paralleli-

sation works correctly and is able to correctly carry out RT simulations when the domain is distributed over multiple cores. The test on a realistic density field for one point source also shows good agreement between `CRASH-AMR` and `PCRASH-AMR` simulations with the latter being run on multiple cores. Regarding the scaling efficiency, we find that the code does not show an optimal performance for both these scenarios where the load is highly imbalanced to begin with. The core containing the single source needs to constantly emit and propagate the photon-packets, the remaining cores will be idle till they have any photon-packets to process. Also, the ray-tracing algorithm has an inherent randomness requiring frequent communication among cores.

Regarding the test cases run on a realistic density field with multiple point sources, we looked at both weak and strong scaling scenarios. The weak scaling tests show fairly good results. On *Odin*, for test cases where the ICs have been set on grids with 3 r.l, we find a scaling efficiency up to 77.6% and 53.2%, respectively, on 64 and 128 cores. This is with RD disabled and keeping 16 cores as the baseline. *Hydra* seems to show a better scaling efficiency of 88.3 and 61.6% on 64 and 128 cores respectively. However, in terms of overall run times, *Odin* performs better than *Hydra*, which can be attributed to the node configuration on *Hydra*. When RD is enabled, the efficiency is 52.1% on *Odin* for 64 cores, which reduces to 41.9% on 128 cores. The corresponding values for *Hydra* are 58.5 and 44.3% respectively.

For the weak scaling tests, with 4 r.l., we find a similar trend. When RD is disabled, the efficiency on *Odin* is 45.6% dropping down to 36% on 64 and 128 cores respectively. The corresponding efficiency on *Hydra* is 51.2 and 42%. With RD enabled, the efficiency is 50 and 39.3% for 64 and 128 cores, respectively, on *Odin*. The corresponding values on Hydra are 56 and 44%. Although the scaling efficiency for test cases with RD on seems to be higher, in terms of run times it is a factor of 1.2 higher than test cases with RD disabled. Here again, *Hydra* has run times higher than *Odin* by a factor of 1.1 on 128 cores. Between test cases with 3 and 4 r.l., we find that the presence of an additional level of refinement causes an increase in run times due to the additional work required to search for the right box and propagate the rays across different levels.

The strong scaling tests also perform fairly well. For the test cases with ICs set on grids with 3 r.l., we find a speed-up of 2.36 and 3.57 on 64 and 128 cores, respectively, for *Odin* when RD is disabled. This is calculated keeping 16 cores as a baseline. The speed-up on *Hydra* for the same tests is 2.63 and 4.1. When RD is enabled, the speed-up on *Odin* is 2.26 and 3.37 for 64 and 128 cores respectively. The corresponding values on *Hydra* are 2.8 and 4.05.

For test cases with 4 r.l., we find a speed-up of 2.53 and 3.74 on 64 and 128 cores, respectively, for *Odin* when RD is disabled. The speed-up on *Hydra* for the same tests is 2.87 and 4.44. For RD enabled, we find a speed-up of 2.72 and 3.92 on *Odin* for 64 and 128 cores. The corresponding values on Hydra are 3.1 and 4.47. In terms of run time, the case with RD enabled performs less efficiently than that with RD disabled, and the total run time is higher by a factor of 1.1.

To conclude, `PCRASH-AMR` can be used to run test cases with multiple sources, on grids with multiple refinement levels, on multiple cores. The code also shows consistent results across different architectures as confirmed by our tests. On the whole we find that the default method of LB in `CHOMBO` performs better than using the RD for balancing the load. The latter results in a more even distribution of load, but with the boxes spread across different

cores. This results in more one-to-one communications among cores to propagate the rays further. Between *Odin* and *Hydra*, we find *Odin* giving a better performance on a larger number of cores. This could be due to our choice of core numbers, where the nodes on *Hydra* are not filled completely. We find that the efficiency and speed-up starts to reduce while moving on to 128 cores. We are considering a number of improvements to the code, that will allow the code to scale up to 512 cores, these are mentioned below.

Further improvements

There are a number of improvements that can be done to the code. The ray-tracing code as of now keeps two maps to search for the right box the ray should propagate to, local and global. The global map as of now contains all the non-local boxes, but it is not necessary to store all this information as we only require a list of boxes that surround a particular core. If we use this information to build the global map, its size will be drastically reduced, thereby reducing the time to search for the right box in the global map. We are currently working on implementing this in PCRASH-AMR.

The LB criteria we have adopted as of now is a static one, done at the beginning of the simulation. Currently, the load on a box does not change during the processing of one snapshot of a cosmological simulation. With multiple snapshots, the density field and other physical quantities will evolve with redshift. This introduces the possibility of additional point sources being added to high density regions in the simulation. At this point, we will require a dynamic LB scheme to redistribute the load among the cores.

Graph partitioning techniques can be looked at as another option to decompose the domain among cores. This can be done, in principle, by assigning weights to boxes at each refinement level depending on whether they are neighbor(s), child(ren) or parent(s) so that they are assigned to the same core. However, as mentioned earlier, domain decomposition in this manner will only work if the geometry of the density field is suitable for it.

Our algorithm for ray propagation does not introduce any delays in communicating the rays when they reach the end of a core domain. This leads to multiple communication events in the same time step, with additional synchronisation events to ensure that all cores are at the same time step. We can consider the possibility of delaying propagation of the photon-packets till the next time step. But it needs to be seen how this would work for refined grids, given that the distance traveled by the ray is not the same at all levels, like in the case of a uniform grid. The time resolution will then have to be adjusted according to the refinement level the ray is traveling in.

All our RT simulations until now have been done in a post-processing mode, using the simulation outputs of a hydro code as input. We would like to consider the possibility of introducing adaptivity within the RT code itself. Doing so provides the advantage that we can refine the domain according to the requirements of the RT simulation instead of the hydro code and focus our computational efforts only where it is necessary. This can be useful in certain situations, for example, predicting the temperature of the gas surrounding a quasar which has important consequences for cosmological studies. We next look at the steps necessary to introduce this adaptivity into CRASH-AMR using CHOMBO.

Chapter 5

Adaptive Radiative Transfer simulations with CRASH

In our previous chapters, we have looked at the methodology adopted to enable RT simulations on AMR grids in CRASH. From the test scenarios it is clear that CRASH-AMR will be able to provide a much better picture of the growth of H II regions through the different spatial scales involved in the reionisation process. Given that the code uses the AMR framework of CHOMBO, which is widely used in the astrophysics community, we can run CRASH-AMR with the outputs of different hydro codes for different setups.

The adaptivity in the AMR grids that we have used for our tests till now is decided by the hydro code. CRASH-AMR uses these grids in a post-processing mode, i.e., we do not carry out any refinements during the RT simulation. In this Chapter we look at the feasibility of implementing an adaptive version of CRASH-AMR, that can refine the grids based on certain criteria specific to RT simulations, using CHOMBO. We discuss the reasons such an adaptivity will be useful in RT simulations. We then give an overview of how CHOMBO can be used to implement adaptive refinement using its AMR framework; finally we look at a sample test case to get more insight into the issues that we need to take into account for introducing adaptivity into CRASH-AMR.

5.1 Adaptivity in Radiative Transfer codes

RT codes in general are used as post-processing tools along with hydro codes that adaptively refine the grids based on various criteria, for example, the overdensity of baryon and dark matter [198], a local density criterion [128] or the gradients of certain variables describing the gas flow that are refined when they exceed a threshold value [184]. The RT code, by itself, does not refine the grid during the simulation. For certain problems, however, it is useful for the RT code to be able to specify certain refinement criteria and refine the grid based on these. One important example would be tracking the I-front moving away from a bright quasar [35]. Being able to resolve the I-front around a quasar allows us to better predict the

temperature of the gas surrounding it; this is important to understand the contribution of quasars in driving the process of IGM reionisation [155, 202].

There are a few RT codes available that implement adaptivity in space and time; we mentioned RADAMESH [35] that uses both the BSAMR and CBAMR framework to propagate the rays across the simulation domain. The code makes use of a "cell by cell" approach, instead of the "ray by ray" approach in CRASH. One important point to be mentioned here is that the code ensures that the grids are strictly nested, i.e. a child grid lies completely within a parent grid. This is different to the PBAMR framework we have used, where a child grid can lie over multiple parent grids. Hence, the approach used in RADAMESH makes it amenable to using a tree structure to store the grid hierarchy. The grids at different levels are connected to the parents and neighbors through linked lists similar to the hierarchy discussed in Section 2.2.1.2.5. A second tree structure is also built, where for each cell that has been refined, it has a pointer or link to the grid that contains its refined cells. The cell based tree allows the code to efficiently determine the grid that the ray passes through while going through higher refinement levels, and the grid based tree is used to find cells at coarser or same refinement levels. The code keeps track of all *active* cells in the domain that represent the region that needs to be refined in the current time step. The refinement criteria used include the cell neutral fraction, the ionisation rate and the cell optical depth. Since the cells that lie within the I-front have the highest ionisation rate they are always selected as being *active*.

FTTE [153] is another example that implements a scheme to do RT on adaptively refined grids. The code solves for radiation transport by assuming a wavefront entering a grid in a direction (ϕ, θ) and considering ray segments in the xy, xz or yz planes. It discretises the rays within each cell, attributing certain properties to each ray within the cell, for example the optical depth, its geometric length and orientation of the cell side that the ray touches first. Each cell itself is part of a grid tree with pointers to all its neighbors in 3D. The intensity within each cell is either set to the cosmic background intensity, or the outgoing intensity of the neighboring cell from which the ray has exited. The code can also take into account ray-splitting, while passing through different refinement levels, using the HEALPix library [1, 72].

The above code examples demonstrate some of the techniques used by RT codes to implement adaptivity. In case of CRASH-AMR, the adaptivity has to be done within CHOMBO, in terms of grid generation and data management, we take a look at how the library implements this adaptivity in its framework.

5.2 Adaptive refinement in CHOMBO

The CHOMBO library, as mentioned in Section 2.2.6 implements the BSAMR framework through a hierarchy of classes with a different functionality. Together these classes provide a framework that can be used to set up adaptivity in a code. We have already looked at some of the classes that are needed to set up the AMR grids in CRASH-AMR, we now look at the classes required to do adaptive refinement.

The main class one has to use in CHOMBO to set up adaptive grids is the *BRMeshRefine* class. This is based on the clustering algorithm developed in [20]. The user can select a set of points

that need to be tagged, and these are used in setting up new grids at a new refinement level. The grids can be defined by using the tagged cells in two different ways; the first is to select the cells only at the base level and create a single set of tags using the *IntVectSet* class. The *BRMeshRefine* class can then appropriately refine the tags at different refinement levels. The second method involves creating an array of tags, consisting of a set of tags for at each level. The class then uses all the tags to define the new grids.

One could also choose to add a new mesh level, or re-define the grids keeping the same number of mesh levels. The new grids created satisfy the criteria for " proper nesting" (Section 2.2.1.3) and are disjoint. Once regridding has been done, a set of new grids are returned which then have to be copied to the respective *DisjointBoxLayout* defined for each level in the *AMRLevel* class. The *FineInterp* and *CoarseAverage* classes can then be used to carry out interpolation or averaging operations between levels. Also the data at the same level is copied onto the new grids using the inbuilt copy classes of CHOMBO.

Now that we know how CHOMBO implements adaptivity, we can look at the feasibility of using this capability in CRASH-AMR.

5.3 Feasibility of adaptive refinement in CRASH-AMR

From the examples of adaptive RT codes mentioned in Section 5.1, we see that these codes follow a cell based approach, whereas CRASH follows a ray based approach. We follow a ray through the grid and at each cell crossing calculate the change in the properties of the medium, for example, the ionisation fractions, T etc. We do not know, a priori, the number of rays that will be crossing through each cell. What we do know is that each cell has to be crossed by a ray a minimum number of times to achieve convergence (§ 3.1.1.1). Also CRASH does not employ a ray splitting scheme, as adopted in codes such as [1, 72], which can be used to split the rays according to the refinement level the ray is traversing through. Additionally, the codes we have mentioned implement the AMR framework within their codes and as such are tailored for their specific requirements. For the case of CRASH-AMR this does not apply. We use CHOMBO to represent our AMR hierarchy, which is not tailored for use with an RT code as such. One could argue that this is somewhat of a overkill in terms of using a library to set up AMR grids in a static manner, but this is not the case.

Currently we use CHOMBO just to set up the AMR hierarchy, but we envisage coupling CRASH with a hydro code in the future. CRASH-AMR is a step forward in providing a code that implements the radiative feedback mechanism in AMR enabled hydro codes. We can readily use the outputs of these hydro codes as input and provide the outputs of the RT simulation back to the hydro code. Also, a number of pre- and post-processing tools that we have set up in CRASH-AMR use the additional classes available in CHOMBO and greatly reduced the time for development efforts. Finally, although we do not refine the grids within CRASH-AMR as of now, we can look at the possibility of doing so, since the code is fully interfaced with the CHOMBO library.

For CRASH-AMR to be able to adaptively refine grids and carry out RT simulations on them, there are two major considerations that need to be taken into account. We need to be able

to decide, during the RT simulation, the cells that need to be refined and how often to refine them. This has more to do with the algorithm used in CRASH. The size of the region selected for refinement should not be so large that it removes any advantage that AMR provides. In the case of CRASH-AMR we need to select the region around the I-front, which can be a few cells thick. The decision of when to refine is also not trivial since the speed with which the I-front propagates is not known a-priori, except for some standard test cases as discussed in [88].

The other consideration has to do with rebuilding the AMR hierarchy itself. The process of point clustering and grid generation during the process of regridding will be taken care of by CHOMBO, but one has to consider the effect that F_r, the fill factor, can have during the process of regridding. Section 2.2.1.2.3 provides more details on how this variable affects the quality of the refined grids generated. The number of grids generated during regridding cannot be so high that the ray tracing algorithm suffers due to having to find a new neighbor, parent or child grid for each cell traversed. This can be the case if F_r is set to a high value. If the value is set too low, we have a smaller number of grids but with cells being refined unnecessarily resulting in large grids at higher resolution; this takes away any benefits of having high resolution only where we need it.

To further understand the implications of doing adaptive RT simulations in CRASH-AMR using CHOMBO, we look at a sample test case.

5.4 Sample test case with CRASH-AMR

In this Section we consider a sample test case, from the point of view of run time performance of regenerating grids in CHOMBO and setting up the new hierarchy in CRASH-AMR. We look at the quality of the grids generated by CHOMBO when we take into account different values for F_r and size of the region being refined.

Consider the set up in Section 3.3.1 with a point source located at the origin of the box. The I-front moving away from this source resembles a shell which is few cells thick. The dynamic refinement has to consider only the cells that lie within this I-front, refine them as the simulation proceeds. Hence, we set up a similar test case and simulate such an I-front by creating a shell of growing size that moves away from the point source as the simulation proceeds. All the cells that fall inside this shell are selected for refinement.

We provide a code snippet below that shows the sequence of operations being done, and at each stage we mention the relevant CHOMBO class that is called.

Listing 5.1 Code snippet for tagging and regridding

```
1
2  !numAMRLevels – number of levels to begin with
3  !maxAMRLevels – maximum number of levels allowed
4
5  CALL setICsOnCHOMBOGrids(numAMRLevels, maxAMRLevels)
6
7  !numRefinements – number of refinements to be done
```

```
8  !numCellsToBeRefined − number of cells to be refined in each loop
9
10 DO index=0, numRefinements − 1
11
12    sphereStartCoords = index ∗ numCellsToBeRefined
13    sphereEndCoords = index ∗ (numCellsToBeRefined + 1)
14
15    CALL simSelectCellsToBeTagged(index)
16
17    CALL simCopyExistingGrids
18
19    CALL simRunRegriddingInCHOMBO
20
21    CALL simSetAMRDataInCRASH
22
23    CALL simWriteOutputs
24
25 END DO
```

The routine `setICsOnCHOMBOGrids` sets up the ICs on the base grid, and calls the required `CHOMBO` routines to set up the AMR framework. At the end of this call, we have the `ChomboAMR` class set up along with pointers to all the refinement levels. Then, depending on the number of times we want to refine the grid, we carry out the following operations.

(a) The routine `simSelectCellsToBeTagged` calls a `CHOMBO` class `Sphere` that selects and tags all the cells that lie within the shell whose radius, R_s, is given by the variable *numCellsToBeRefined*. The start and end coordinates of the sphere are calculated in the variables *sphereStartCoords* and *sphereEndCoords*. We tag cells only at the base grid level, these are then used by the `BRMeshRefine` class and refined where ever necessary to create the new grid levels.

(b) Before the regrid can be done, the routine `simCopyExistingGrids` backs up the boxes in `DisjointBoxLayout` and `FArrayBox` containing the data at each level.

(c) Then, the routine `simRunRegriddingInCHOMBO` calls the `regrid` routine within the `BRMeshRefine` class that creates a new level as required. If a new refinement level is created we set up the `DisjointBoxLayout` and the data in the `FArrayBox` for this level.

(d) The `simSetAMRDataInCRASH` clears the existing data structure in `CRASH-AMR` and sets up the new AMR hierarchy. This includes determining the *localID* and *globalID* of each box and whether each cell is refined or not. The neighbor, parent and child lists are updated and copied to `CRASH-AMR`. In summary, the complete data structure as discussed in Section 3.2.2 is set up.

(e) Finally, the routine `simWriteOutputs` writes out the data in HDF5 format.

We have run the above test case for different values of R_s and F_r. We start with a single grid at the base level, and go up to a maximum of three refinements, i.e. four levels in the grid hierarchy. The variable *numRefinements* is set to 10 and during each refinement the shell being refined moves away from the grid origin. This essentially means that the number of cells that are selected for refinement grows larger.

Figure 5.1 shows the total run time (in seconds) for carrying out the regridding (*regrid*) and post-regridding (*postRegrid*) within CHOMBO and the rebuilding of the hierarchy in CRASH-AMR (*getGridInfo*), for different values of F_r and R_s. The *regrid* phase consists of calling the BRMeshRefine class, *postRegrid* sets up the relevant data that is passed back to CRASH-AMR. Here we determine the neighbor(s), parent(s) and child(ren) of each box and assign its unique *localID* and *globalID*. Finally, during *getGridInfo* we set up the Fortran data structure in CRASH-AMR and get the pointers to the data from the *FArrayBox*.

From the graphs it is clear that the *regrid* operation takes longer for $R_s > 5$ cells and $F_r > 0.75$ and increases by a factor of 1.2 between $F_r = 0.7$ and 0.85. Regarding the number of cells refined, the time taken to regrid increases by a factor of 8.5 when $R_s = 10$, when compared to $R_s = 5$. For the *postRegrid* operation, we find that when $F_r = 0.7$, the time taken for the test case where $R_s = 10$ cells is 23 times higher when compared to $R_s = 5$. However, as the value of F_r reaches 0.8 or more, the run time for $R_s = 10$ cells is higher by a factor of 8 when compared to $R_s = 5$. For the *getGridInfo* operation, when $F_r = 0.7$, the difference in run times for different values of R_s is almost negligible. However, when F_r is set to 0.8 and 0.85, the difference in run times between the test cases with $R_s = 10$ and $R_s = 5$ is a factor of 1.5 and 8, respectively.

Essentially, we find that the performance of the *regrid* and *getGridInfo* operations degrades with an increase R_s and F_r. As we move away from the origin, the number of cells that fall within a shell grows larger and results in a large number of boxes being created to cover the tagged cells. The *postRegrid* operation seems to be able to carry out the copy operations efficiently and does not show any reduction in performance unlike the other two operations.

Note that we have done 10 refinements in total, and during the last refinement stage, if $R_s = 10$, then the values for *sphereStartCoords* and *sphereEndCoords* are 90 and 99. Given that the base grid resolution in this case is 128^3, we have covered only $\sim 46\%$ in volume of the box for refinement. From the standard test case in Section 3.3.1, for a box of size $L = 6.6$ kpc and 128^3 resolution, the Strömgren sphere extends up to 5.4 kpc, which covers about $\sim 53\%$ in volume of the box. So the time taken for refining the grid will be higher. The standard test in Section 3.3.1 takes approximately 60 minutes to complete, however introducing adaptivity could increase the run times since the code has to now solve the RT equations on a larger number of cells that lie within the refined region and also carry out the relevant operations in CHOMBO. Although the regrid operation that lasts $\sim$ 10 - 11 minutes will constitute a smaller proportion of the run time when adaptivity is introduced, we still have to carefully set up the refinement criteria, i.e. R_s, F_r and the number of times we want to do a refinement. This will ensure that we can carry out the RT simulation in a feasible time frame and not get bogged down by the time taken to dynamically refine the grid.

5.5 Conclusion

We have considered a sample test case to look at the feasibility of implementing adaptivity in CRASH-AMR. Given that the framework necessary to carry out the tagging and regridding operations is in place, we should be able to set this up in CRASH-AMR albeit with certain careful considerations. Introducing adaptivity in CRASH-AMR will require many changes to the

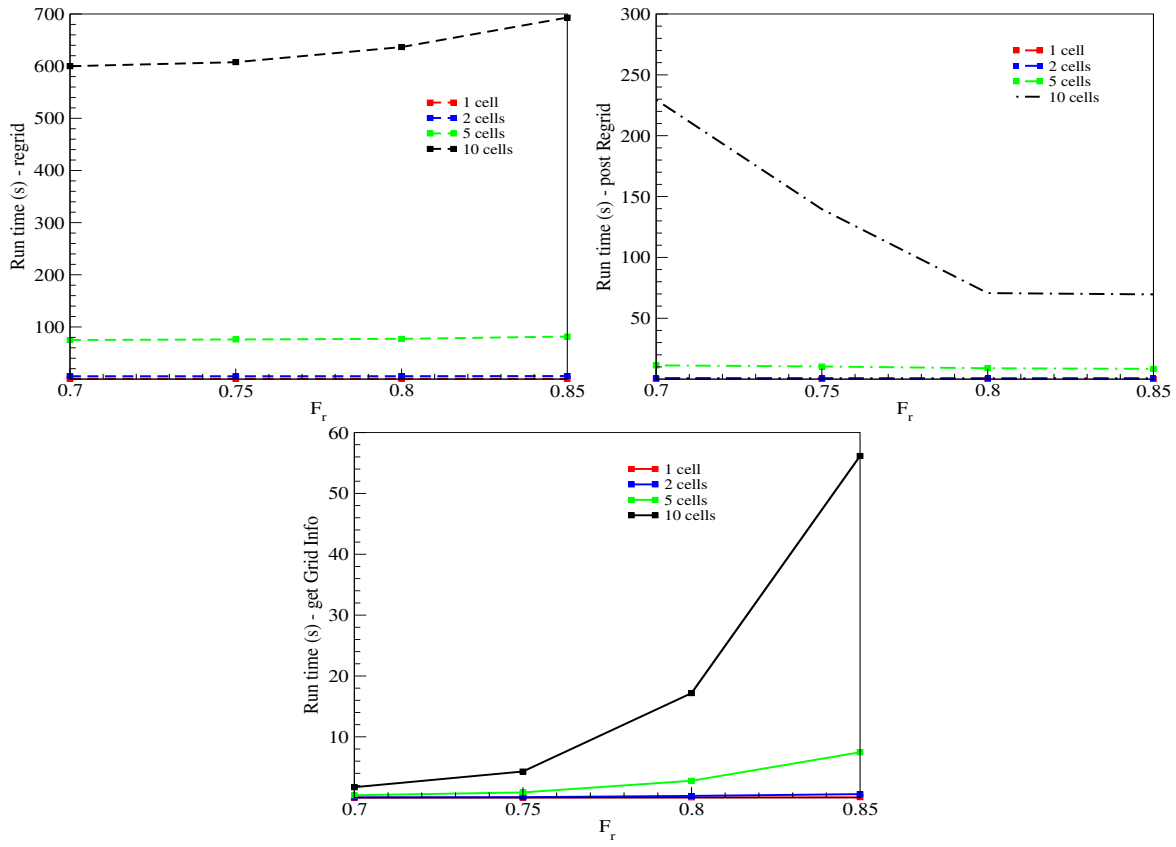

Figure 5.1 Run time (s) for different operations to regrid and rebuild the AMR hierarchy, for different values of F_r in `CRASH-AMR`. The different colors refer to the number of cells in the sphere radius selected for refinement, 1 cell (red), 2 cells (blue), 5 cells (green) and 10 cells (black). **Top:** Run times for regrid and post regrid operations in `CHOMBO`. **Bottom:** Run times for rebuilding the hierarchy in `CRASH-AMR`

.

code in terms of being able to decide, accurately, the region of interest and frequency with which the grids should be refined. The code follows a "ray by ray" approach which makes it non-trivial to decide a-priori where the I-front will be at a certain point. We could make some assumptions about the speed of the I-front and refine the grid such that the I-front always stays within this refined region, but this may not be the ideal scenario and could result in cells being unnecessarily refined. The situation becomes more complex in a cosmological scenario, with multiple point sources. The ionised bubble moving away from a point source might merge with that from other point sources and require further refinements. However, to begin with, being able to resolve the I-front around a single point source is of much importance in the cosmological scenario and we plan to look at this in detail in future.

Chapter 6

Conclusion

High resolution simulations are a crucial requirement for understanding the various physical processes involved in cosmology. The development of efficient algorithmic and numerical methods is thus necessary to make judicious use of the computational resources available. Radiative transfer (RT) simulations provide an important tool to understand the effects of radiative feedback on structure formation and still present a unique set of computational challenges.

In this thesis we have looked at improving the computational and algorithmic capabilities of the RT code `CRASH`, thus enabling it to run RT simulations on static, refined grids thereby focusing the computational efforts on regions of interest. For this purpose, the code has been interfaced with the open-source AMR library `CHOMBO`, which provides the necessary framework required to store and manage the grid hierarchy. The library is not originally intended for a code like `CRASH` which requires calculations to be done on a cell-by-cell basis to simulate radiation-matter interaction. Nevertheless, we have been successful in adopting the library for our purposes and have implemented a complete and stable interface between `CRASH` and `CHOMBO`. A detailed discussion of the new code and the development methodology, following the software architecture of the baseline code `CRASH3`, has been done. We have also taken advantage of the various tools available in the library to provide a number of pre- and post-processing tools for use with `CRASH-AMR`. These enable a user to set-up simple, standard, test cases and also to analyse the outputs of `CRASH-AMR` simulations.

We have then carried out a series of tests, both with the simplified set-up prescribed in the RT Code Comparison Project (RTCCP) and a realistic hydrodynamic simulation with AMR refinement. All the standard tests show a good agreement with the latest release of `CRASH`, thus confirming that the interface between `CRASH-AMR` and `CHOMBO` is correct. It also proves that `CRASH-AMR` is able to perform consistent RT simulations on a more accurate representation of the gas distribution in the cosmological domain. Our tests done on a realistic density field show a sharp difference in the ionised region patterns formed at different AMR resolutions; this is due to the more accurate treatment of the radiation-matter interaction.

`CRASH-AMR` provides a number of computational and algorithmic improvements over `CRASH`. The code is now able to perform RT simulations with the resolution of the density field

increased by a factor of 64, which was previously impossible on CRASH due to memory limitations of a single compute core. When compared with RT simulations run on high-resolution uniform grids, CRASH-AMR gives a 59% reduction in terms of the total run time.

Our next step has been to parallelise CRASH-AMR using distributed memory parallelism, this has provided a new set of challenges due to the inherent nature of the ray-tracing algorithm in the code. The rays emitted from point sources travel in random directions, requiring frequent communication among cores. The presence of refined grids distributed across multiple cores adds to the complexity of the code. We discuss the implementation of the parallel code PCRASH-AMR in detail, providing an insight into the different techniques used to parallelise the code. Two load balancing schemes have been discussed along with the methodology adopted to map the distributed grid hierarchy available in CHOMBO back onto CRASH in an efficient way. The improvements to the ray-tracing algorithm, that takes advantage of the SAMR framework to optimise ray propagation to other cores, have been mentioned. PCRASH-AMR has been tested with a standard test case prescribed in RTCCP and we find a good agreement with CRASH-AMR. To study the performance of the code, weak and strong scaling tests have been done, on two different machines, for grids with three and four refinement levels. We have been able to get fairly good, if not perfect, scaling with 128 cores. A number of immediate improvements has been suggested; we are currently working on some of them. This should improve the scaling efficiency of the code to 256 - 512 cores. This development will now allow us to carry out high-resolution cosmological simulations on larger problem sizes involving thousands of point sources.

CRASH-AMR is as of now used as a post-processing tool for adaptive hydro codes; the RT simulation does not carry out any refinements. Adaptivity within the RT code is extremely useful for certain problems, for example resolving the temperature changes around a quasar. This is of importance in cosmological scenarios, for example the IGM reionisation. The final part of this thesis deals with the possibility of introducing adaptivity into CRASH-AMR using the SAMR framework of CHOMBO. We discuss the necessary changes required to do so and look at a simple test case to understand the various algorithmic issues involved. The interface developed between CRASH-AMR and CHOMBO requires certain extensions for this; a number of changes that need to be done to the RT algorithm in CRASH have been discussed.

The developments carried out as part of this thesis now make it possible to carry out high-resolution RT simulations with CRASH; as a result of this work we can now look at the feasibility of coupling the code with a cosmological hydrodynamic code thereby enabling self-consistent, large scale simulations of structure formation.

Bibliography

[1] T. Abel and B. D. Wandelt. Adaptive ray tracing for radiative transfer around point sources. *MNRAS*, 330:53–56, March 2002. 2.2.4, 5.1, 5.3

[2] M. Adams et al. CHOMBO Software Package for AMR Applications - Design Document. *Lawrence Berkeley National Laboratory Technical Report LBNL-6616E*, 2011. 2.2.5, 2.18, 3

[3] M. Ainsworth and J. T. Oden. *A Posteriori Error Estimation in Finite Element Analysis*. John Wiley& Sons, 2000. 2.2.2.1, a

[4] A. S. Almgren et al. CASTRO: A New Compressible Astrophysical Solver. I. Hydrodynamics and Self-gravity. *ApJ*, 715:1221–1238, June 2010. 2.2.1.3

[5] M. A. Alvarez, V. Bromm, and P. R. Shapiro. The H II Region of the First Star. *ApJ*, 639:621–632, March 2006. 2.2.4

[6] R. E. Angulo et al. Scaling relations for galaxy clusters in the Millennium-XXL simulation. *MNRAS*, 426:2046–2062, November 2012. 1.3, 2.2.4

[7] Babuska, I. and Strouboulis, T. and Upadhyay, C.S. A model study of the quality of a posteriori error estimators for linear elliptic problems. Error estimation in the interior of patchwise uniform grids of triangles. *Computer Methods in Applied Mechanics and Engineering*, 114(3-4):307 – 378, 1994. ISSN 0045-7825. 2.2.2.1

[8] M. Bader. Texts in computational science and engineering. In *Space Filling Curves*, volume 9 of Texts in Computational Science and Engineering, 2013. 4.1.1

[9] E. Baensch. Local mesh refinement in 2 and 3 dimensions. {*IMPACT*} *of Computing in Science and Engineering*, 3(3):181 – 191, 1991. ISSN 0899-8248. 2.2.2.2

[10] M. Baes, H. Dejonghe, and J. I. Davies. Efficient radiative transfer modelling with SKIRT. In *The Spectral Energy Distributions of Gas-Rich Galaxies: Confronting Models with Data*, volume 761 of *American Institute of Physics Conference Series*, pages 27–38, April 2005. 2.2.4

[11] S. Balay et al. PETSc Users Manual. Technical Report ANL-95/11 - Revision 3.5, Argonne National Laboratory, 2014. 2.2.5

[12] R. A. Bank, H. A. Sherman, and A. Weiser. Some Refinement Algorithms And Data Structures For Regular Local Mesh Refinement, 1983. 2.2.2.2

[13] R. Barkana and A. Loeb. The physics and early history of the intergalactic medium. *Reports on Progress in Physics*, 70:627–657, April 2007. 1.2

[14] T. J. Barth, A. S. Wiltberger, and A. S. Gandhi. Three-dimensional unstructured grid generation via an incremental insertion and local optimization. *NASA Conference Publication*, 3143:449–461, 1992. 2.1.2.2

[15] J. Bédorf, E. Gaburov, and S. Portegies Zwart. Bonsai: N-body GPU tree-code. Astrophysics Source Code Library, December 2012. 1.3

[16] J. Bédorf et al. 24.77 Pflops on a Gravitational Tree-Code to Simulate the Milky Way Galaxy with 18600 GPUs. *ArXiv e-prints*, December 2014. 1.3

[17] C. L. Bennett et al. Nine-year Wilkinson Microwave Anisotropy Probe (WMAP) Observations: Final Maps and Results. *APJS*, 208:20, October 2013. 1.2

[18] J. M Berger and J. R. Leveque. Adaptive mesh refinement using wave-propagation algorithms for hyperbolic systems. SIAM J. Numer. Anal, 35:2298–2316, 1998. 1.1, 2.2.5

[19] J. M. Berger and J. Oliger. Adaptive mesh refinement for hyperbolic partial differential equations. Journal of Computational Physics, 53(3):484 – 512, 1984. ISSN 0021-9991. 1.1, 2.2, 2.2.1.2, 2.2.1.2, 2.2.6

[20] J. M. Berger and I. Rigoutsos. An algorithm for point clustering and grid generation. IEEE Transactions on Systems, Man and Cybernetics, 21(5):1278–1286, sep/oct 1991. ISSN 0018-9472. doi: 10.1109/21.120081. 2.2.1.2.3, 2.13, 5.2

[21] D. C. Black and P. Bodenheimer. Evolution of rotating interstellar clouds. II - The collapse of protostars of 1, 2, and 5 solar masses. *APJ*, 206:138–149, May 1976. 1.3

[22] D. T. Blacker and J. R. Meyers. Seams and wedges in plastering: A 3-D hexahedral mesh generation algorithm. *Engineering with Computers*, 9(2):83–93, 1993. ISSN 0177-0667. 2.1.2.1

[23] D. T. Blacker and B. M. Stephenson. Paving: A new approach to automated quadrilateral mesh generation. *"International Journal for Numerical Methods in Engineering"*, 32(4):811–847, 1991. ISSN 1097-0207. 2.1.2.1

[24] E. Blayo and L. Debreu. Adaptive mesh refinement for finite-difference ocean models: first experiments. *Journal of Physical Oceanography*, 29(6):1239–1250, 1999. 2.2.1

[25] J. Bonet and J. Peraire. An alternating digital tree (ADT) algorithm for 3D geometric searching and intersection problems. *International Journal for Numerical Methods in Engineering*, 31(1):1–17, 1991. ISSN 1097-0207. 2.1.2.1

[26] K. Borne. Astroinformatics: A 21st Century Approach to Astronomy Research and Education. In *American Astronomical Society Meeting Abstracts, 215*, volume 42 of *Bulletin of the American Astronomical Society*, page 230.01, January 2010. 1.3

[27] A. Bowyer. Computing Dirichlet tessellations. *The Computer Journal*, 24(2):162–166, 1981. 2.1.2.2

[28] M. Boylan-Kolchin, V. Springel, S. D. M. White, A. Jenkins, and G. Lemson. Resolving cosmic structure formation with the Millennium-II Simulation. *MNRAS*, 398:1150–1164, September 2009. 2.2.4

[29] C. Brinch and M. R. Hogerheijde. LIME - a flexible, non-LTE line excitation and radiation transfer method for millimeter and far-infrared wavelengths. *AAP*, 523:A25, November 2010. 2.2.4

[30] G. L. Bryan et al. Enzo: An Adaptive Mesh Refinement Code for Astrophysics. *ArXiv e-prints*, July 2013. 1.3, 2.2.4

[31] Greg L. Bryan. Fluids in the universe: Adaptive mesh refinement in cosmology. *Computing in Science and Engg.*, 1(2):46–53, March 1999. ISSN 1521-9615. 2.2.4

[32] L. Buntemeyer et al. Radiation Hydrodynamics using Characteristics on Adaptive Decomposed Domains for Massively Parallel Star Formation Simulations. *ArXiv e-prints*, January 2015. 4.1.1, 4.2.2.1

[33] C. Burstedde et al. Scalable adaptive mantle convection simulation on petascale supercomputers. In *Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, page 62. IEEE Press, 2008. 2.2.3

[34] P. Camps, M. Baes, and W. Saftly. Using 3D Voronoi grids in radiative transfer simulations. *AAP*, 560:A35, December 2013. 2.2.4

[35] S. Cantalupo and C. Porciani. RADAMESH: cosmological radiative transfer for Adaptive Mesh Refinement simulations. *MNRAS*, 411:1678–1694, March 2011. 2.2.1.3, 2.2.4, 5.1

[36] R. G. Carlberg and H. M. P. Couchman. Mergers and bias in a cold dark matter cosmology. *APJ*, 340:47–68, May 1989. 1.3

[37] Q. Chen and M. Gunzburger. Goal-oriented a Posteriori Error Estimation for Finite Volume Methods. *J. Comput. Appl. Math.*, 265:69–82, August 2014. ISSN 0377-0427. 2.2.2.1

[38] B. Ciardi. Cosmic reionization and the LOFAR project. In *Cosmic Radiation Fields: Sources in the early Universe (CRF 2010)*, page 34, 2010. 1.2, 2.2.4

[39] B. Ciardi and A. Ferrara. The First Cosmic Structures and Their Effects. *SSR*, 116: 625–705, February 2005. 1.2

[40] B. Ciardi, A. Ferrara, S. Marri, and G. Raimondo. Cosmological reionization around the first stars: Monte Carlo radiative transfer. *MNRAS*, 324:381–388, June 2001. aiii, 3.1

[41] B. Ciardi, A. Ferrara, and S. D. M. White. Early reionization by the first galaxies. *MNRAS*, 344:L7–L11, September 2003. 3

[42] B. Ciardi, F. Stoehr, and S. D. M. White. Simulating intergalactic medium reionization. *MNRAS*, 343:1101–1109, August 2003. 3

[43] B. Ciardi, J. S. Bolton, A. Maselli, and L. Graziani. The effect of intergalactic helium on hydrogen reionization: implications for the sources of ionizing photons at z ¿ 6. *MNRAS*, 423:558–574, June 2012. 3

[44] P. Colella and P. R. Woodward. The Piecewise Parabolic Method (PPM) for Gas-Dynamical Simulations. *Journal of Computational Physics*, 54:174–201, September 1984. 1.1

[45] M. Compostella, S. Cantalupo, and C. Porciani. The imprint of inhomogeneous He II reionization on the H I and He II Lyα forest. *MNRAS*, 435:3169–3190, November 2013. 1.2

[46] J. W. Cooley and John W. Tukey. An algorithm for the machine calculation of complex fourier series. *Mathematics of Computation*, 19(90):pp. 297–301, 1965. ISSN 00255718. 1.1

[47] R. Courant, K. Friedrichs, and H. Lewy. On the partial difference equations of mathematical physics. *IBM Journal of Research and Development*, 11(2):215–234, march 1967. ISSN 0018-8646. doi: 10.1147/rd.112.0215. 2.2.1.1.2

[48] R. A. Crain et al. Galaxies-intergalactic medium interaction calculation - I. Galaxy formation as a function of large-scale environment. *MNRAS*, 399:1773–1794, November 2009. 1.3

[49] A. J. Cunningham et al. Simulating Magnetohydrodynamical Flow with Constrained Transport and Adaptive Mesh Refinement: Algorithms and Tests of the AstroBEAR Code. *ApJS*, 182:519–542, June 2009. 2.2.5

[50] W. W. Dai. Issues in adaptive mesh refinement. In IEEE International Symposium on Parallel Distributed Processing, Workshops and Phd Forum (IPDPSW),, pages 1 –8, april 2010. doi: 10.1109/IPDPSW.2010.5470758. 2.2.1.3

[51] R. Davé. Simulations of the Intergalactic Medium. In *Maps of the Cosmos*, volume 216 of *IAU Symposium*, page 251, January 2005. 1.2

[52] R. Deiterding. Construction and Application of an AMR Algorithm for Distributed Memory Computers. In Tomasz Plewa, Timur Linde, and V. Gregory Weirs, editors, *Adaptive Mesh Refinement - Theory and Applications*, volume 41 of *Lecture Notes in Computational Science and Engineering*, pages 361–372. Springer Berlin Heidelberg, 2005. ISBN 978-3-540-21147-1. 4.1

[53] Ralf Deiterding. Detonation structure simulation with AMROC. In Performance Computing and Communications,, pages 916–927. Springer, 2005. 2.2.5

[54] B. Delaunay. Sur la sphère vide. *Bull. Acad. Science USSR VII: Class Sci. Mat. Nat.*, 7:793, 1934. 2.1.2.2

[55] Demkowicz, L. and Devloo, Ph. and Oden, J.T. On an h-type mesh-refinement strategy based on minimization of interpolation errors. *Computer Methods in Applied Mechanics and Engineering*, 53(1):67 – 89, 1985. ISSN 0045-7825. 2.2.2.1

[56] Diaz, R. A. and Kikuchi, N. and Taylor, E. J. A method of grid optimization for finite element methods. *Computer Methods in Applied Mechanics and Engineering*, 41(1):29 – 45, 1983. ISSN 0045-7825. d

[57] E. Dorfi. 3D models for self-gravitating, rotating magnetic interstellar clouds. *AAP*, 114:151–164, October 1982. 1.3

[58] P. C. Duffell and A. I. MacFadyen. TESS: A Relativistic Hydrodynamics Code on a Moving Voronoi Mesh. *APJS*, 197:15, December 2011. 2.2.4

[59] G. Efstathiou and J. W. Eastwood. On the clustering of particles in an expanding universe. *MNRAS*, 194:503–525, February 1981. 1.3

[60] L. Freitag, M. Jones, and P. Plassmann. Mesh component design and software integration within sumaa3d, 1999. 2.2.5

[61] C. S. Frenk et al. The Santa Barbara Cluster Comparison Project: A Comparison of Cosmological Hydrodynamics Solutions. *APJ*, 525:554–582, November 1999. 3.3.3

[62] B. Fryxell et al. FLASH: An Adaptive Mesh Hydrodynamics Code for Modeling Astrophysical Thermonuclear Flashes. *ApJS*, 131:273–334, November 2000. 1.3, 2.2.4, 2.2.6

[63] Alan J. G. *Computer Implementation of the Finite Element Method*. PhD thesis, PhD Thesis, 1971, Stanford, CA, USA, 1971. AAI7205916. 2.1.2.1

[64] V. N. Gamezo et al. Thermonuclear supernovae: Simulations of the deflagration stage and their implications. *Science*, 299(5603):77–81, 2003. 2.2.4

[65] C. W. Gear and R. D. Skeel. A history of scientific computing. chapter The Development of ODE Methods: A Symbiosis Between Hardware and Numerical Analysis, pages 88–105. ACM, New York, NY, USA, 1990. ISBN 0-201-50814-1. 1.1

[66] J. M. Gelb and E. Bertschinger. Cold dark matter. 1: The formation of dark halos. *APJ*, 436:467–490, December 1994. 1.3

[67] R. A. Gingold and J. J. Monaghan. Smoothed particle hydrodynamics - Theory and application to non-spherical stars. *MNRAS*, 181:375–389, November 1977. 1.1

[68] M. Gittings et al. The RAGE radiation-hydrodynamic code. Computational Science and Discovery, 1(1):015005, October 2008. 2.2.4

[69] N. Y. Gnedin and T. Abel. Multi-dimensional cosmological radiative transfer with a Variable Eddington Tensor formalism. *New Astronomy*, 6:437–455, October 2001. b, 2.2.4

[70] N. Y. Gnedin and J. P. Ostriker. Reionization of the Universe and the Early Production of Metals. *APJ*, 486:581–598, September 1997. 1.3.1

[71] N. Y. Gnedin, K. Tassis, and A. V. Kravtsov. Modeling Molecular Hydrogen and Star Formation in Cosmological Simulations. *APJ*, 697:55–67, May 2009. 2.2.4

[72] K. M. Górski et al. HEALPix: A Framework for High-Resolution Discretization and Fast Analysis of Data Distributed on the Sphere. *ApJ*, 622:759–771, April 2005. 2.2.4, 5.1, 5.3

[73] L. Graziani, A. Maselli, and B. Ciardi. crash3: cosmological radiative transfer through metals. *MNRAS*, 431(1):722–740, 2013. 3.1, 4

[74] S. L. Grimm and J. G. Stadel. The GENGA Code: Gravitational Encounters in N-body Simulations with GPU Acceleration. *APJ*, 796:23, November 2014. 1.3

[75] C.P.T. Groth et al. A parallel adaptive 3d mhd scheme for modeling coronal and solar wind plasma flows. *"Space Science Reviews"*, 87:193–198, 1999. ISSN 0038-6308. doi: 10.1023/A:1005136115563. 2.2.4

[76] E. J. Groth and P. J. E. Peebles. Statistical analysis of catalogs of extragalactic objects. VII - Two- and three-point correlation functions for the high-resolution Shane-Wirtanen catalog of galaxies. *APJ*, 217:385–405, October 1977. 1.3

[77] S. Habib et al. HACC: Simulating Sky Surveys on State-of-the-Art Supercomputing Architectures. *ArXiv e-prints*, October 2014. 1.3

[78] A. Hannukainen, S. Korotov, and M. Krizek. On global and local mesh refinements by a generalized conforming bisection algorithm. *Journal of Computational and Applied Mathematics*, 235(2):419 – 436, 2010. ISSN 0377-0427. Special Issue on Advanced Computational Algorithms. 2.2.2.2

[79] J. Harnois-Déraps et al. High-performance P^3M N-body code: CUBEP^{3}M. *MNRAS*, 436:540–559, November 2013. 2.2.4

[80] K. Heng. The Nature of Scientific Proof in the Age of Simulations. *ArXiv e-prints*, April 2014. 1.3

[81] M. Heroux, P. Raghavan, and H. Simon. *Parallel Processing for Scientific Computing*. Society for Industrial and Applied Mathematics, 2006. doi: 10.1137/1.9780898718133. 2.2.3

[82] M. R. Hestenes. A history of scientific computing. chapter Conjugacy and Gradients, pages 167–179. ACM, New York, NY, USA, 1990. ISBN 0-201-50814-1. 1.1

[83] P. Hut et al. Smooth Particle Hydrodynamics: Models, Applications, and Enabling Technologies. *ArXiv Astrophysics e-prints*, October 1997. 1.3

[84] I. Iben, Jr. Stellar Evolution. I. The Approach to the Main Sequence. *APJ*, 141:993, April 1965. 1.3

[85] I. Iben, Jr. Stellar Evolution. II. The Evolution of a 3 M_{sun} Star from the Main Sequence Through Core Helium Burning. *APJ*, 142:1447, November 1965. 1.3

[86] I. T. Iliev et al. Simulating cosmic reionization at large scales - I. The geometry of reionization. *MNRAS*, 369:1625–1638, July 2006. 1.2

[87] I. T. Iliev et al. Cosmological radiative transfer codes comparison project - I. The static density field tests. *MNRAS*, 371:1057–1086, September 2006. 1.3.1, 2.2.4, 3, 3.3.1, 3.3.1.1, 3.3.1.2, 3.3.2, 3.3.2.2

[88] I. T. Iliev et al. Cosmological radiative transfer comparison project - II. The radiation-hydrodynamic tests. *MNRAS*, 400:1283–1316, December 2009. 2.2.4, 5.3

[89] T. Ishiyama, T. Fukushige, and J. Makino. GreeM: Massively Parallel TreePM Code for Large Cosmological N -body Simulations. *PASJ*, 61:1319–, December 2009. 1.3

[90] Jasak, H. and Gosman, A.D. Element residual error estimate for the finite volume method. *Computers & Fluids*, 32(2):223 – 248, 2003. ISSN 0045-7930. a

[91] H. Johansson. Design and implementation of a dynamic and adaptive meta-partitioner for parallel SAMR grid hierarchies. 2008. 4.1

[92] H. Johansson and A. Vakili. A patch-based partitioner for parallel SAMR applications. 2008. 4.1

[93] T. M. Jones and E. P. Plassmann. Adaptive refinement of unstructured finite-element meshes, 1997. 2.2.2.2, 2.17

[94] T. A. Karl et al. Using RngStreams for parallel random number generation in C++ and R. *Computational Statistics*, 29(5):1301–1320, 2014. ISSN 0943-4062. 4.2.5

[95] George Karypis and Vipin Kumar. A Parallel Algorithm for Multilevel Graph Partitioning and Sparse Matrix Ordering. *Journal of Parallel and Distributed Computing*, 48(1):71 – 95, 1998. ISSN 0743-7315. 2.2.5

[96] A. Khokhlov. Fully Threaded Tree Algorithms for Adaptive Refinement Fluid Dynamics Simulations. Journal of Computational Physics, 143:519–543, July 1998. 2.1, 2.2, 2.2.1.1.1, 2.2.1.1.2, 2.2.1.1.2, 2.2.4

[97] J. Kim et al. The New Horizon Run Cosmological N-Body Simulations. *Journal of Korean Astronomical Society*, 44:217–234, December 2011. 2.2.4

[98] B. S. Kirk, J. W. Peterson, R. H. Stogner, and G. F. Carey. `libMesh`: A C++ Library for Parallel Adaptive Mesh Refinement/Coarsening Simulations. *Engineering with Computers*, 22(3–4):237–254, 2006. 2.2.5

[99] R. I. Klein, R. T. Fisher, C. F. McKee, and M. R. Krumholz. Recent Advances in the Collapse and Fragmentation of Turbulent Molecular Cloud Cores. In *Star Formation in the Interstellar Medium: In Honor of David Hollenbach*, volume 323 of *Astronomical Society of the Pacific Conference Series*, page 227, December 2004. 2.2.4

[100] D. E. Knuth. *The Art of Computer Programming, Volume 2 (3rd Ed.): Seminumerical Algorithms*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1997. ISBN 0-201-89684-2. 4.2.5

[101] J. Kondo. *Supercomputing applications, algorithms, and architectures for the future of supercomputing*. Springer-Verlag Tokyo, 1991. 1.1

[102] A. V. Kravtsov, A. A. Klypin, and A. M. Khokhlov. Adaptive Refinement Tree: A New High-Resolution N-Body Code for Cosmological Simulations. *ApJS*, 111:73, July 1997. 2.2.4

[103] M. Kuhlen, M. Vogelsberger, and R. Angulo. Numerical simulations of the dark universe: State of the art and the next decade. *Physics of the Dark Universe*, 1:50–93, November 2012. 2.2.4

[104] Z. Lan, V. E. Taylor, and G. Bryan. A novel dynamic load balancing scheme for parallel systems. *Journal of Parallel and Distributed Computing*, 62(12):1763–1781, 2002. 4.1

[105] Z. Lan, V. E Taylor, and G. Bryan. A novel dynamic load balancing scheme for parallel systems. *Journal of Parallel and Distributed Computing*, 62(12):1763–1781, 2002. 4.1

[106] B. Laney, C. *Computational Gasdynamics*. Cambridge University Press, 1998. 1.1

[107] A. M. C. Le Brun et al. Towards a realistic population of simulated galaxy groups and clusters. *MNRAS*, 441:1270–1290, June 2014. 1.3

[108] P. L'Ecuyer et al. An object-oriented random-number package with many long streams and substreams. *Operations Research*, 50:1073–1075, 2002. 4.2.5

[109] C. D. Levermore. Relating Eddington factors to flux limiters. *JQSRT*, 31:149–160, February 1984. 2.2.4

[110] Li, X. and Shephard, S. M. and Beall, M. W. 3D anisotropic mesh adaptation by mesh modification. *Computer Methods in Applied Mechanics and Engineering*, 194(48-49): 4915 – 4950, 2005. ISSN 0045-7825. Unstructured Mesh Generation. 2.2.3

[111] S. Lin and B. W. Kernighan. An effective heuristic algorithm for the traveling-salesman problem. Operations Research, 21(2):498–516, 1973. 2.2.6.1

[112] Anwei Liu and Barry Joe. On the shape of tetrahedra from bisection. *Mathematics of Computation*, 63(207):141–154, 1994. 2.2.2.2

[113] R. Loehner and P. Parikh. Generation of three-dimensional unstructured grids by the advancing-front method. *International Journal for Numerical Methods in Fluids*, 8(10): 1135–1149, 1988. ISSN 1097-0363. 2.1.2.1

[114] J. Z. Lou et al. "a robust and scalable library for parallel adaptive mesh refinement on unstructured meshes". In *Solving Irregularly Structured Problems in Parallel*, volume 1457 of *Lecture Notes in Computer Science*, pages 156–169. Springer Berlin Heidelberg, 1998. ISBN 978-3-540-64809-3. 2.2.5

[115] L. B. Lucy. A numerical approach to the testing of the fission hypothesis. *AJ*, 82: 1013–1024, December 1977. 1.1

[116] T. Lunttila and M. Juvela. Radiative transfer on hierarchial grids. *AAP*, 544:A52, August 2012. 2.2.4

[117] M. N. Machida et al. Collapse and fragmentation of rotating magnetized clouds - II. Binary formation and fragmentation of first cores. *MNRAS*, 362:382–402, September 2005. 1.3

[118] J. Mackey, V. Bromm, and L. Hernquist. Three Epochs of Star Formation in the High-Redshift Universe. *APJ*, 586:1–11, March 2003. 1.2

[119] P. MacNeice et al. Paramesh: A parallel adaptive mesh refinement community toolkit. Computer Physics Communications, 126(3):330 – 354, 2000. ISSN 0010-4655. 2.2.5

[120] G. Marsaglia. The Marsaglia random number CDROM including the DIEHARD battery of tests of randomness, 1995. *http://www.stat.fsu.edu/pub/diehard*, 2008. 4.2.5

[121] M. Mascagni and A. Srinivasan. Algorithm 806: SPRNG: A scalable library for pseudorandom number generation. *ACM Transactions on Mathematical Software (TOMS)*, 26(3):436–461, 2000. 4.2.5

[122] A. Maselli and A. Ferrara. Radiative transfer effects on the Lyα forest. *MNRAS*, 364: 1429–1440, December 2005. 3, 3.1

[123] A. Maselli, B. Ciardi, and A. Kanekar. CRASH2: coloured packets and other updates. *MNRAS*, 393:171–178, February 2009. 3.1

[124] A. Maselli et al. CRASH: a radiative transfer scheme. *MNRAS*, 345:379–394, October 2003. 3.1, 3.1.1, 3.1.1.1

[125] D. J. Mavriplis. Unstructured grid techniques. *Annual Review of Fluid Mechanics*, 29 (1):473–514, 1997. 2.1

[126] A. A. Meiksin. The physics of the intergalactic medium. *Reviews of Modern Physics*, 81:1405–1469, October 2009. 1.2

[127] A. Mignone et al. The PLUTO Code for Adaptive Mesh Computations in Astrophysical Fluid Dynamics. *ApJS*, 198:7, January 2012. 1.3, 2.2.4, 2.2.6

[128] F. Miniati and P. Colella. Block structured adaptive mesh and time refinement for hybrid, hyperbolic + N-body systems. Journal of Computational Physics, 227:400–430, November 2007. 1.3, 2.2, 2.2.4, 2.2.6, 2.2.6.2, 3.3, 3.3.3, 5.1

[129] K. Miyoshi and T. Kihara. Development of the correlation of galaxies in an expanding universe. *PASJ*, 27:333–346, 1975. 1.3

[130] A. Moore. *Dynamical Simulations of Extrasolar Planetary Systems with Debris Disks Using a GPU Accelerated N-Body Code*. PhD thesis, University of Rochester, 2013. 1.3

[131] G. M. Morton. A computer Oriented Geodetic Data Base; and a New Technique in File Sequencing. *Technical Report, Ottawa, Canada: IBM Ltd*, 1966. 2.2.6.1

[132] S. G. Nash, editor. *A History of Scientific Computing*. ACM, New York, NY, USA, 1990. ISBN 0-201-50814-1. 1.1

[133] C. J. J. Nellenback et al. Efficient parallelization for {AMR} {MHD} multiphysics calculations; implementation in astrobear. *Journal of Computational Physics*, 236(0): 461 – 476, 2013. ISSN 0021-9991. 4.1

[134] Nordstroem, J., Forsberg, K., Adamsson, C. Eliasson, P. Finite volume methods, unstructured meshes and strict stability for hyperbolic problems. *Applied Numerical Mathematics*, 45(4):453 – 473, 2003. ISSN 0168-9274. 2.1

[135] M. L. Norman. Historical perspective on astrophysical MHD simulations. In *Computational Star Formation*, volume 270 of *IAU Symposium*, pages 7–17, April 2011. 1.3

[136] C.D. Norton, J.Z. Lou, and T. Cwik. Status and directions for the pyramid parallel unstructured amr library. In *Parallel and Distributed Processing Symposium., Proceedings 15th International*, pages 1224–1231, April 2001. 2.2.5

[137] J. T. Oden. Historical comments on finite elements. In *A history of scientific computing*, pages 152–166. ACM, 1990. 1.1

[138] E. C. Olson et al. Electronic computations of astrophysical interest. *AJ*, 63:52, February 1958. 1.3

[139] Steven J. Owen, Matthew L. Staten, Scott A. Canann, and Sunil Saigal. "advancing front quadrilateral meshing using triangle transformations", 1998. 2.1.2.1

[140] J.-P. Paardekooper, V. Icke, and J. Ritzerveld. Radiation Hydrodynamics of First Stars with SimpleX Radiative Transfer. In *First Stars III*, volume 990 of *American Institute of Physics Conference Series*, pages 453–455, March 2008. 2.2.4

[141] J.-P. Paardekooper, C. J. H. Kruip, and V. Icke. SimpleX2: radiative transfer on an unstructured, dynamic grid. *AAP*, 515:A79, June 2010. 2.2.4

[142] M. Parashar and J. C. Browne. An infrastructure for parallel adaptive mesh refinement techniques. Technical report, University of Texas, Austin, 1995. a, 4.1

[143] M. Parashar and J. C. Browne. On partitioning dynamic adaptive grid hierarchies. In *System Sciences, 1996., Proceedings of the Twenty-Ninth Hawaii International Conference on,*, volume 1, pages 604–613. IEEE, 1996. 4.1

[144] A. M. Partl, A. Maselli, B. Ciardi, A. Ferrara, and V. Müller. Enabling parallel computing in CRASH. *MNRAS*, 414:428–444, June 2011. 1.3.1, 3.1, 4, 4.1.1, 4.2.1, 4.2.3.2

[145] P. J. E. Peebles. The Nature of the Distribution of Galaxies. *AAP*, 32:197, May 1974. 1.3

[146] M. Pierleoni, A. Maselli, and B. Ciardi. CRASHα: coupling continuum and line radiative transfer. *MNRAS*, 393:872–884, March 2009. 3.1

[147] Planck Collaboration. Planck 2015 results. I. Overview of products and scientific results. *ArXiv e-prints*, February 2015. 1.2

[148] C. S. Plesko et al. Hydrocode Modeling of Asteroid Impacts into a Volatile-Rich Martian Surface: Initial Results,. In AAS/Division for Planetary Sciences Meeting Abstracts, volume 37 of Bulletin of the American Astronomical Society, page 691, August 2005. 2.2.4

[149] T. Plewa, L. Timur, and V. W. Gregory. AMR - Theory and Applications. In *Proceedings of the Chicago Workshop on Adaptive Mesh Refinement Methods*, volume 41 of Lecture Notes in Computational Science and Engineering, September 2003. 2.2.1.2.1, 2.2.4

[150] D. Price. *Smoothed Particle Hydrodynamics*. PhD thesis, PhD Thesis, 2005, July 2005. 2.2.4

[151] J. Rantakokko. Partitioning strategies for structured multiblock grids. *Parallel Computing*, 26(12):1661–1680, 2000. 4.1

[152] J. Rantakokko and M. Thuné. Parallel structured adaptive mesh refinement. In *Parallel computing*, pages 147–173. Springer, 2009. 4.1

[153] A. O. Razoumov and C. Y. Cardall. Fully threaded transport engine: new method for multi-scale radiative transfer. *MNRAS*, 362:1413–1417, October 2005. 2.2.4, 5.1

[154] E.-J. Rijkhorst et al. Hybrid characteristics: 3d radiative transfer for parallel adaptive mesh refinement hydrodynamics. *A&A*, 452:907–920, June 2006. ai, aii, 2.2.1.3, 4.1.1

[155] E. Ripamonti, M. Mapelli, and S. Zaroubi. Radiation from early black holes - I. Effects on the neutral intergalactic medium. *MNRAS*, 387:158–172, June 2008. 5.1

[156] C. M. Rivara. Mesh Refinement Processes Based on the Generalized Bisection of Simplices. *SIAM Journal on Numerical Analysis*, 21(3):604–613, 1984. 2.17, 2.2.2.2, 2.2.2.2

[157] C. M. Rivara. Selective refinement/derefinement algorithms for sequences of nested triangulations. *International Journal for Numerical Methods in Engineering*, 28:2889–2906, 1989. 2.2.2.2

[158] C. M. Rivara and C. Levin. A 3-D refinement algorithm suitable for adaptive and multi-grid techniques. *Communications in Applied Numerical Methods*, 8(5):281–290, 1992. ISSN 1555-2047. 2.2.2.2

[159] J. Rosdahl, J. Blaizot, D. Aubert, T. Stranex, and R. Teyssier. RAMSES-RT: radiation hydrodynamics in the cosmological context. *MNRAS*, 436:2188–2231, December 2013. 1.3, 2.2.4

[160] H. Samet. Neighbour finding in images represented by octrees. In *Computer Vision, Graphics and Image Processing*, pages 367–386, 1989. 2.2.1.1.1

[161] E. Scannapieco, R. J. Thacker, and M. Davis. High-Redshift Galaxy Outflows and the Formation of Dwarf Galaxies. *APJ*, 557:605–615, August 2001. 1.2

[162] M. Schaefer. *Computational Engineering - Introduction to Numerical methods*. Springer, 2006. ISBN 3-540-30685-4. 2.1, 2.1.1.2

[163] J. Schaye et al. The physics driving the cosmic star formation history. *MNRAS*, 402: 1536–1560, March 2010. 1.3

[164] J. Schaye et al. The EAGLE project: Simulating the evolution and assembly of galaxies and their environments. *ArXiv e-prints*, July 2014. 1.3, 2.2.4

[165] R. Schneider et al. Low-mass relics of early star formation. *Nature*, 422:869–871, April 2003. 1.2

[166] L. D. Shaw et al. Statistics of Physical Properties of Dark Matter Clusters. *ApJ*, 646: 815–833, August 2006. 2.2.4

[167] M. Shee, S. Bhavsar, and M. Parashar. Characterizing the performance of dynamic distribution and load-balancing techniques for adaptive grid hierarchies. In *Proceedings IASTED International conference of parallel and distributed computing and systems*, 1999. 4.1

[168] S. Shweta and M. Parashar. Adaptive Runtime Partitioning of AMR Applications on Heterogeneous Clusters, 2001. 4.1

[169] C. M. Simpson et al. The Effect of Feedback and Reionization on Star Formation in Low-mass Dwarf Galaxy Halos. In *American Astronomical Society Meeting Abstracts*, volume 221 of *American Astronomical Society Meeting Abstracts*, page 107.05, January 2013. 2.2.4

[170] Smith, E. R. and Eriksson, E. L. Algebraic grid generation. *Computer Methods in Applied Mechanics and Engineering*, 64(1-3):285 – 300, 1987. ISSN 0045-7825. 2.1.1.1

[171] V. Springel. Smoothed particle hydrodynamics in astrophysics. *A&A*, 48(1):391–430, 2010. 2.2.4

[172] V. Springel. Moving-mesh hydrodynamics with the AREPO code. In Computational Star Formation, volume 270 of IAU Symposium, pages 203–206, April 2011. 2.2.4

[173] V. Springel and L. Hernquist. Cosmological smoothed particle hydrodynamics simulations: a hybrid multiphase model for star formation. *MNRAS*, 339:289–311, February 2003. 1.2

[174] V. Springel, C. S. Frenk, and S. D. M. White. The large-scale structure of the Universe. *Nature*, 440:1137–1144, April 2006. 1.2

[175] R. Spurzem et al. Supermassive Black Hole Binaries in High Performance Massively Parallel Direct N-body Simulations on Large GPU Clusters. In R. Capuzzo-Dolcetta, M. Limongi, and A. Tornambè, editors, *Advances in Computational Astrophysics: Methods, Tools, and Outcome*, volume 453 of *Astronomical Society of the Pacific Conference Series*, page 223, July 2012. 1.3

[176] J. Steensland. Dynamic structured grid hierarchy partitioners using inverse space-filling curves. *Technical Report200*, pages 1–00, 2001. 4.1

[177] J. Steensland, S. Chandra, and M. Parashar. An application-centric characterization of domain-based SFC partitioners for parallel SAMR. *Parallel and Distributed Systems, IEEE Transactions on*, 13(12):1275–1289, 2002. 4.1

[178] J. Steensland et al. Towards an adaptive meta-partitioner for parallel SAMR applications. In *Proceedings of the IASTED International Conference on Parallel and Distributed Computing Systems, Las Vegas*, pages 425–430, 2000. 4.1

[179] Johan Steensland. Efficient partitioning of dynamic structured grid hierarchies. 2002. 4.1

[180] T. Suginohara et al. Cosmological N-body simulations with a tree code - Fluctuations in the linear and nonlinear regimes. *APJS*, 75:631–643, March 1991. 1.3

[181] Y. Suto. Simulations of Large-Scale Structure in the New Millennium. In M. Colless, L. Staveley-Smith, and R. A. Stathakis, editors, *Maps of the Cosmos*, volume 216 of *IAU Symposium*, page 105, January 2005. 1.3

[182] A. B. Szabo. Mesh design for the p-version of the finite element method. *Computer Methods in Applied Mechanics and Engineering*, 55(1-2):181 – 197, 1986. ISSN 0045-7825. c

[183] Masaharu Tanemura, Tohru Ogawa, and Naofumi Ogita. A new algorithm for three-dimensional voronoi tessellation. *Journal of Computational Physics*, 51(2):191 – 207, 1983. ISSN 0021-9991. 2.1.2.2

[184] R. Teyssier. Cosmological hydrodynamics with adaptive mesh refinement. A new high resolution code called RAMSES. *A&A*, 385:337–364, April 2002. 1.3, 2.2, 2.2.1.3, 5.1

[185] The Planck Collaboration. The Scientific Programme of Planck. *ArXiv Astrophysics e-prints*, April 2006. 1.2

[186] F. J. Thompson, K. B. Soni, and P. N. Weatherill. *Handbook of Grid Generation*. CRC Press, 1999. ISBN 0849326877. 2.1.1.1, 2.1.1.1.1

[187] Michael Thuné. Partitioning strategies for composite grids. *Parallel Algorithms and Applications*, 11(3-4):325–348, 1997. 4.1

[188] H. Trac and R. Cen. Radiative Transfer Simulations of Cosmic Reionization With Pop II and III Stars. In *First Stars III*, volume 990 of *American Institute of Physics Conference Series*, pages 445–449, March 2008. 2.2.4

[189] B. van Straalen et al. Scalability challenges for massively parallel AMR applications. In IEEE International Symposium on Parallel Distributed Processing, IPDPS 2009,, pages 1–12, may 2009. 2.2.6.1

[190] B. van Straalen et al. Petascale block-structured AMR applications without distributed meta-data. In Proceedings of the 17th international conference on Parallel processing - Volume Part II,, *Euro-Par'11*, pages 377–386, Berlin, Heidelberg, 2011. Springer-Verlag. ISBN 978-3-642-23396-8. 2.2.6.1

[191] M. Vestias and H. Neto. Trends of cpu, gpu and fpga for high-performance computing. In *Field Programmable Logic and Applications (FPL), 2014 24th International Conference on*, pages 1–6, Sept 2014. 1.1

[192] M. Vogelsberger et al. Introducing the Illustris Project: simulating the coevolution of dark and visible matter in the Universe. *MNRAS*, 444:1518–1547, October 2014. 1.3

[193] G. Voronoi. Nouvelles applications des paramètres continus à la théorie des formes quadratiques. Premier mémoire: Sur quelques proprieteés des formes quadratiques positives parfaites. *J. Reine Angew. Math.*, 134:198, 1908. 2.1.2.2

[194] M. S. Warren. 2HOT: An Improved Parallel Hashed Oct-Tree N-Body Algorithm for Cosmological Simulation. *ArXiv e-prints*, October 2013. 1.3

[195] D. F. Watson. Computing the n-dimensional Delaunay tessellation with application to Voronoi polytopes. *The Computer Journal*, 24(2):167–172, 1981. 2.1.2.2, 2.1.2.2

[196] S. D. M. White et al. Clusters, filaments, and voids in a universe dominated by cold dark matter. *APJ*, 313:505–516, February 1987. doi: 10.1086/164990. 1.3

[197] B. H. Wilde et al. "Three-dimensional RAGE Simulations of Strong Shocks Interacting with Sapphire Balls". In APS Meeting Abstracts,, page 8096P, November 2007. 2.2.4

[198] J. H. Wise and T. Abel. ENZO+MORAY: radiation hydrodynamics adaptive mesh refinement simulations with adaptive ray tracing. *MNRAS*, 414:3458–3491, July 2011. 1.3, 1.3.1, 1.3.1, 2.2, 2.2.4, 4.1.1, 4.2.1, 4.2.2.1, 5.1

[199] M. A. Wissink et al. Large scale parallel structured AMR calculations using the SAMRAI framework. In SC01 Conference on High Performance Networking and Computing,, 2001. 2.2.5

[200] G. Worseck et al. The End of Helium Reionization at z ˜= 2.7 Inferred from Cosmic Variance in HST/COS He II Lyα Absorption Spectra. *APJL*, 733:L24, June 2011. 1.2

[201] K. K. S. Wu, O. Lahav, and M. J. Rees. The large-scale smoothness of the Universe. *Nature*, 397:225–230, January 1999. 1.2

[202] J. S. B. Wyithe and A. Loeb. Reionization of Hydrogen and Helium by Early Stars and Quasars. *APJ*, 586:693–708, April 2003. 5.1

[203] Z. Xu, J. Glimm, and X. Li. Front tracking algorithm using adaptively refined meshes. In *Adaptive Mesh Refinement - Theory and Applications*, volume 41 of *Lecture Notes in Computational Science and Engineering*, pages 83–89. Springer Berlin Heidelberg, 2005. ISBN 978-3-540-21147-1. 2.2

[204] P. D. Young et al. "a locally refined rectangular grid finite element method: Application to computational fluid dynamics and computational physics". Journal of Computational Physics, 92(1):1–66, 1991. ISSN 0021-9991. 2.2.1.1

[205] J. Z. Zhu, O. C. Zienkiewicz, E. Hinton, and J. Wu. A new approach to the development of automatic quadrilateral mesh generation. *"International Journal for Numerical Methods in Engineering"*, 32(4):849–866, 1991. ISSN 1097-0207. 2.1.2.1

[206] U. Ziegler and H. W. Yorke. A nested grid refinement technique for magnetohydrodynamical flows. *Computer Physics Communications*, 101:54–74, April 1997. 1.3

[207] O. C. Zienkiewicz and J. Z. Zhu. A simple error estimator and adaptive procedure for practical engineerng analysis. *International Journal for Numerical Methods in Engineering*, 24(2):337–357, 1987. ISSN 1097-0207. b

[208] D. Zuzio and J. L. Estivalezes. An efficient block parallel AMR method for two phase interfacial flow simulations. *"Computers & Fluids"*, 44(1):339–357, 2011. ISSN 0045-7930. 2.2.1.2