Alma Mater Studiorum · Università di Bologna

DIPARTIMENTO DI INFORMATICA - SCIENZA E INGEGNERIA
Corso di Laurea in Ingegneria e Scienze Informatiche

ACCELERARE I RAGGI COSMICI SULLA GPU

Relatore:
Chiar.mo Prof.
Moreno Marzolla

Presentata da:
Alessandro Ronchi

Correlatore:
Chiar.mo Prof.
Claudio Gheller

Sessione 27/11/2025
Anno Accademico 2024/2025

Sommario

I raggi cosmici sono particelle ad alta energia scoperte un centinaio di anni fa, e tutt'ora si è incerti sulla loro origine esatta. Lo studio di queste particelle tramite le sole osservazioni astronomiche è complicato, in quanto la loro sorgente non può essere tracciata in modo diretto, e i dati a nostra disposizione non hanno ancora una qualità sufficiente per dare una conferma definitiva ai modelli teorici che descrivono la loro formazione. Per questo motivo, in parallelo ad osservazioni astronomiche sempre più accurate, recentemente si è affermata la disciplina dell'astrofisica computazionale, il cui obiettivo è utilizzare gli strumenti e le tecnologie offerte dall'informatica per ricreare i fenomeni astronomici mediante simulazioni di vario tipo. Nello specifico, per lo studio dei raggi cosmici si utilizzano simulazioni idrodinamiche che riproducono il comportamento dei cluster di galassie, consentendo di osservare la loro evoluzione nel tempo. A partire da queste simulazioni è possibile ricavare dei dati che possono essere direttamente confrontati con le osservazioni astronomiche reali, in modo da verificare la validità dei modelli teorici elaborati. Spesso, le simulazioni di questo tipo comportano tempi di calcolo molto elevati, dovuti al gran numero di computazioni effettuate al loro interno, e ciò rende più complicato attuare gli approcci sperimentali che ne prevedono l'utilizzo. Per far fronte a questo problema, si ricorre a tecniche di high performance computing, che consentono di sfruttare il parallelismo offerto dalle architetture hardware moderne per migliorare le prestazioni dei programmi utilizzati, riducendo notevolmente i tempi di esecuzione.

In tale contesto, all'interno di questa tesi si considera un programma che

consente di generare delle osservazioni sui raggi cosmici a partire dai risultati di una simulazione idrodinamica svolta in precedenza. In particolare, il programma considerato implementa il metodo di Chang-Cooper che consente di risolvere computazionalmente l'equazione di Fokker-Planck per il calcolo dello spettro di emissione delle particelle della simulazione. In questo programma è già presente un tipo di parallelismo che utilizza le risorse hardware della CPU (Central Processing Unit) per ridurre il tempo di calcolo. L'obiettivo della tesi è valutare l'efficienza del parallelismo CPU già implementato nel codice, misurando le prestazioni attuali, per poi realizzare una versione del programma che sfrutta l'elevata capacità computazionale delle GPU (Graphics Processing Unit). Quest'ultima versione è realizzata facendo uso della libreria OpenMP, che utilizza un approccio ad alto livello basato sulle direttive e consente di produrre codice portabile su più architetture differenti. Si intende dunque valutare l'efficacia e la facilità di utilizzo di OpenMP nell'ottimizzare sulla GPU un codice di grandi dimensioni. Il lavoro presentato in questa tesi prevede quindi una prima fase di analisi delle caratteristiche del codice di riferimento, seguita da una valutazione delle prestazioni allo stato attuale, e infine una fase di implementazione del codice GPU basata sull'approccio a direttive offerto da OpenMP. Le ultime due fasi sono state svolte sfruttando l'architettura ad alte prestazioni del supercomputer Leonardo, messo a disposizione dal CINECA. Al termine di questa tesi, si considerano vantaggi e svantaggi dell'utilizzo di OpenMP rispetto a tecnologie più consolidate per il calcolo parallelo sulla GPU, quali CUDA, OpenCL, ROCm e OpenACC, e si illustrano i possibili sviluppi futuri legati alla versione GPU realizzata.

Indice

Sommario **i**

1 Introduzione **1**

1.1 I raggi cosmici e la loro origine 2

1.2 Osservazioni astronomiche 5

1.2.1 Osservazioni dirette 6

1.2.2 Osservazioni indirette 6

1.3 Simulare i raggi cosmici 8

1.3.1 L'equazione di Fokker-Planck per il calcolo dello spettro 11

1.3.2 Il metodo di Chang-Cooper 13

2 Codice di riferimento **17**

2.1 Caratteristiche 17

2.1.1 Input 18

2.1.2 Output 18

2.1.3 Algoritmo e complessità 20

2.1.4 Parallelismo 22

2.2 Test iniziali 23

2.3 Valutazione delle prestazioni del codice CPU 26

2.3.1 Speedup e strong scaling efficiency 27

2.3.2 Weak scaling efficiency 30

2.3.3 Considerazioni 31

3 Versione GPU **33**
3.1 Architettura di una GPU 33
3.2 Programmazione GPU con OpenMP 35
3.2.1 Gerarchia di parallelismo 36
3.2.2 Mapping dei dati 37
3.3 Strategia di parallelismo adottata 38
3.4 Implementazione 40
3.4.1 Caratteristiche della versione ridotta 40
3.4.2 Inserimento delle direttive 42
3.4.3 Refactoring del codice 47
3.4.4 Compilazione 50
3.5 Esecuzione del codice e debugging 51

4 Conclusioni **55**
4.1 Considerazioni sui risultati ottenuti 55
4.2 Sviluppi futuri 57
4.2.1 Valutazione delle prestazioni del codice CPU 57
4.2.2 Versione GPU 57

A Lavorare nell'ambiente di Leonardo **61**
A.1 Cenni sull'architettura di Leonardo 61
A.2 Utilizzo del file system 62
A.3 Scheduler e creazione di job 64
A.4 Gestione dei moduli 67
A.5 Consumo delle risorse 68

Bibliografia **72**

Capitolo 1

Introduzione

I raggi cosmici sono particelle che si muovono nello spazio interstellare a velocità molto elevate, prossime a quella della luce. Queste particelle ricoprono un ruolo fondamentale nell'evoluzione delle galassie, in quanto, lungo il loro percorso, interagiscono continuamente con i vari corpi celesti che le compongono. Il loro studio ci consente quindi di far luce su diversi interrogativi riguardo i fenomeni che accadono nell'Universo e gli oggetti in essi coinvolti. Negli anni, sono state proposte diverse teorie riguardo gli oggetti astronomici che possono essere in grado di produrre i raggi cosmici: dal nostro Sole, ai resti di supernova, o "supernova remnants", fino ai buchi neri supermassivi che si trovano al centro delle galassie. Per cercare di verificare queste teorie e studiare i fenomeni ad esse correlati, sono stati realizzati diversi tipi di strumenti di osservazione, come telescopi e satelliti, che hanno consentito di osservare i raggi cosmici sia direttamente, grazie alla rivelazione delle particelle che li compongono, sia indirettamente attraverso le particelle secondarie e le radiazioni da essi prodotte. Le sole osservazioni, tuttavia, non consentono ancora di ottenere una quantità di dati sufficiente ad analizzare tali fenomeni in modo esaustivo. Inoltre, la verifica sperimentale dei modelli che descrivono la formazione dei raggi cosmici non può essere effettuata in laboratorio, in quanto i processi analizzati coinvolgono oggetti astronomici di grandi dimensioni, come i cluster di galassie, il cui comportamento non

è replicabile su scale ridotte. Questo è il motivo per cui, oltre a cercare di ottenere osservazioni di qualità sempre più elevata, si ricorre a simulazioni che consentono di ricreare i fenomeni ritenuti responsabili della formazione dei raggi cosmici.

1.1 I raggi cosmici e la loro origine

Le particelle che compongono i raggi cosmici sono di varia natura. La maggior parte dei raggi cosmici è formata da nuclei di atomi altamente ionizzati e carichi positivamente. Molti di questi sono semplici protoni provenienti da atomi di idrogeno, ma in percentuali minori si hanno anche nuclei più pesanti, dall'elio fino al piombo. Meno comunemente, tra i raggi cosmici si osservano anche altri tipi di particelle subatomiche, come gli elettroni, aventi carica negativa, e i positroni. L'energia di queste particelle, espressa in elettronvolt (eV), è generalmente molto elevata, ma può variare in un intervallo molto ampio, che va da decine di MeV fino a quasi un ZeV [6].

Sin dalla loro scoperta, si è cercato di identificare le possibili origini dei raggi cosmici. Già da tempo, il nostro Sole è stato identificato come fonte di raggi cosmici ad energia relativamente ridotta, pari ad un centinaio di MeV. Queste particelle sono denominate SAP (Solar Energetic Particles), e vengono rilasciate nel corso delle tempeste solari. Non è ancora del tutto chiaro, invece, come vengano prodotti i raggi cosmici con livelli di energia più elevati, dell'ordine dei GeV o PeV, che si originano al di fuori del nostro sistema solare e prendono il nome di "raggi cosmici galattici". Una volta rilasciati dalla loro fonte, i raggi cosmici galattici viaggiano nello spazio interstellare, che ha una densità di materia molto bassa, ed essendo carichi vengono deviati dai campi magnetici al suo interno. L'influenza del campo magnetico sulla loro traiettoria rende difficile risalire in modo diretto all'origine di queste particelle, ed è uno dei motivi per cui tutt'ora si è incerti sui fenomeni responsabili della loro formazione. Le teorie più accreditate fanno risalire l'origine dei raggi cosmici a due tipi di oggetti che si trovano nell'Universo:

i resti di supernova, o "supernova remnants" [1], e i buchi neri supermassivi che si trovano al centro delle galassie, detti anche AGN (Active Galactic Nuclei) [4]. Entrambi, infatti, soddisfano i requisiti di energia necessari per la produzione dei raggi cosmici, ma mentre i supernova remnants riescono a produrre particelle con energia pari al massimo ad un PeV, gli AGN sono in grado di emettere protoni ed elettroni aventi un'energia ancora più elevata.

Il comportamento di questi oggetti astronomici può essere modellato con tre processi principali:

- Collasso Gravitazionale.
- Shock Acceleration.
- Emissione di radiazione di sincrotrone.

Il Collasso Gravitazionale è un processo che converte l'energia gravitazionale in energia termica e cinetica, provocando l'aggregazione di oggetti di piccole dimensioni in strutture aventi una massa e un'energia più elevata. In particolare, durante questo processo, la forza gravitazionale che agisce su un oggetto astronomico di grandi dimensioni ne provoca la compressione verso il proprio centro di gravità. Il collasso gravitazionale è fondamentale nell'evoluzione dell'Universo, ed è responsabile, tra le altre cose, della formazione di supernovae e buchi neri. Inoltre, può portare all'emissione di grandi quantità di energia sotto forma di "shock", delle "onde d'urto" che si muovono nello spazio interstellare, e si osservano anche in corrispondenza di supernovae e AGN.

La Shock Acceleration è il processo che porta all'accelerazione di ioni ed elettroni, a partire dall'energia generata durante uno shock. In particolare, durante lo spostamento di uno shock nello spazio interstellare, le particelle che si trovano al suo interno attraversano ripetutamente i campi magnetici che si formano in corrispondenza del piano dello shock, subendo un'accelerazione. Queste particelle si muovono da una parte all'altra del piano dello shock con oscillazioni sempre più consistenti, e quando la quantità di energia

in esse contenute è sufficiente, vengono espulse dalla regione di accelerazione e diventano raggi cosmici.

L'emissione di radiazione di sincrotrone, infine, descrive il fenomeno per cui una particella carica con energia relativistica che si trova in un campo magnetico emette una certa quantità di radiazioni. Queste radiazioni possono essere emesse in una varietà di lunghezze d'onda, tra cui quelle di onde radio, raggi X e raggi gamma. L'insieme di frequenze in cui vengono emesse tali radiazioni costituisce lo spettro di emissione della particella.

Per quanto riguarda le supernovae, esse si formano quando stelle di massa molto elevata raggiungono la fine del loro ciclo vitale. Solitamente, ciò accade una volta che la stella è arrivata a produrre, all'interno del suo nucleo, degli atomi di ferro. Il ferro, infatti, è uno degli elementi più stabili nell'universo, e la sua fusione richiede una quantità di energia più elevata di quella che sprigiona. Dunque, una volta che il nucleo della stella contiene una quantità elevata di ferro, la reazione termonucleare si ferma, e la stella subisce un collasso gravitazionale, in cui tutto il materiale di cui è composta viene attirato verso il nucleo. Questo evento dà origine alla supernova, un'esplosione che provoca la formazione di shock nella regione che circonda la stella. Gli shock generati costituiscono i supernova remnants, che continuano a muoversi nello spazio interstellare.

I buchi neri supermassivi che costituiscono gli AGN si formano in un processo simile a quello delle supernovae, in cui il collasso gravitazionale provoca la concentrazione di una massa enorme in un volume molto piccolo. La forte attrazione gravitazionale di questi buchi neri fa sì che essi siano circondati da un'elevata quantità di materiale, che si dispone in strutture dette dischi di accrescimento. Parte del materiale viene, inoltre, espulso dagli AGN in getti che si muovono lungo gli assi di rotazione dei dischi. È in corrispondenza di questi dischi e getti che si ritiene che le particelle vengano accelerate a velocità estremamente elevate, fino a raggiungere livelli di energia superiori ad un PeV.

1.2 Osservazioni astronomiche

La scoperta dei raggi cosmici risale al 1912, quando Victor Hess osservò, tramite una serie di esperimenti condotti con dei palloni aerostatici, che il livello di radiazioni ionizzanti rilevate nell'atmosfera aumenta man mano che si sale di altitudine, e non diminuisce né durante la notte, né durante un'eclissi solare. Tale risultato dimostrava che il livello di radiazioni nell'atmosfera terrestre è dovuto non solo alla radioattività della Terra o all'attività del Sole, ma anche ad una sorgente che si trova al di fuori del nostro Sistema Solare. Da allora, sono state condotte numerose osservazioni con l'obiettivo di analizzare le caratteristiche dei raggi cosmici e risalire alla loro origine. Lo scopo delle osservazioni è ricavare una serie di dati utili per lo studio di queste particelle, tra cui:

- La loro composizione.
- Lo spettro di energia.
- La direzione di provenienza.

Lo spettro di energia descrive, per ogni livello energetico, il flusso di raggi cosmici, cioè il numero di particelle che transitano in una certa area nell'unità di tempo. Questa misura ci consente di capire la quantità di raggi cosmici che si possono rilevare in un arco di tempo, a seconda della loro energia.

La maggior parte delle osservazioni che si effettuano per lo studio dei raggi cosmici è basata sulla rilevazione delle radiazioni emesse da queste particelle quando si trovano all'interno di un campo magnetico. Le osservazioni di questo tipo consentono di osservare gli oggetti astronomici su diverse lunghezze d'onda, in particolare quelle della luce visibile e quelle di onde radio, raggi X e raggi gamma, e pur non concentrandosi direttamente sui raggi cosmici, sono fondamentali per studiare i fenomeni che li producono. Queste osservazioni sono inoltre complementate da tecniche che consentono di rilevare i raggi cosmici nel momento in cui interagiscono con la Terra. Le tecniche di rilevazione dei raggi cosmici sono distinte in due classi:

- Osservazioni dirette.
- Osservazioni indirette.

1.2.1 Osservazioni dirette

Le osservazioni dirette sfruttano le interazioni dei raggi cosmici all'interno dei rivelatori di particelle, come gli spettrometri e i calorimetri. Questi strumenti hanno la necessità di interagire con i raggi cosmici prima che essi entrino in contatto con l'atmosfera, e per questo motivo le osservazioni dirette sono solitamente condotte tramite esperimenti posti su satelliti, stazioni spaziali orbitanti come la ISS (International Space Station), o palloni aerostatici che raggiungono altitudini molto elevate. Gli esperimenti di questo tipo sono soggetti a diversi vincoli tecnici, legati al fatto che gli oggetti inviati in orbita o caricati su palloni aerostatici non possono avere un volume e una massa troppo eccessivi. Ciò limita la superficie di rivelazione a disposizione degli strumenti utilizzati, impedendo di rilevare i raggi cosmici con flusso ridotto. Infatti, minore è il flusso dei raggi cosmici ad una certa energia, maggiore è la superficie di rivelazione necessaria per individuarli. In generale, il flusso dei raggi cosmici diminuisce all'aumentare della loro energia, e di conseguenza le osservazioni dirette consentono di osservare solo le particelle con un'energia minore di 100 TeV, che giungono sulla Terra in maggiori quantità. I raggi cosmici osservati al di sotto di questo livello energetico provengono dalla Via Lattea, e quindi sono detti di origine galattica.

1.2.2 Osservazioni indirette

Quando i raggi cosmici entrano nell'atmosfera terrestre, collidono con i nuclei di aria al suo interno, generando sciami di particelle secondarie che prendono il nome di Extensive Air Showers (EAS). Le EAS sono costituite da milioni o miliardi di particelle che si muovono "a cascata" nell'atmosfera e si formano a partire da una singola particella primaria. In particolare, quando un raggio cosmico collide con un nucleo di aria, l'interazione genera fino a

diverse centinaia di particelle secondarie. Ognuna di queste può poi decadere in un altro tipo di particella oppure continuare ad interagire, dando origine ad ulteriori particelle che contribuiscono a loro volta alla crescita esponenziale dell'EAS. Tra i tipi di particelle che si formano in questo processo si hanno pioni, muoni, adroni e neutrini. In Figura 1.1 è mostrato uno schema che illustra questo processo.

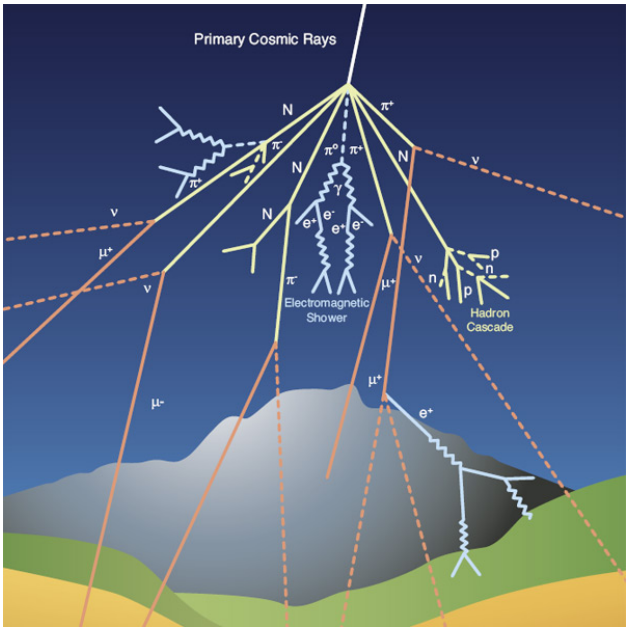

Figura 1.1: Schema che mostra la formazione di un EAS a partire da una particella primaria. Fonte: [5].

Le osservazioni indirette si basano sulla rilevazione di queste particelle secondarie mediante strumenti come i rivelatori di Cherenkov e gli scintillatori. Queste osservazioni sono condotte in esperimenti effettuati sulla superficie terrestre oppure in laboratori sotterranei, e consentono di rilevare anche i raggi cosmici a più alta energia, come gli UHECRs (Ultra-High Energy Cosmic Rays), aventi un'energia superiore ad un EeV. Infatti, in questi esperimenti il flusso ridotto dei raggi cosmici più energetici è compensato dall'utilizzo di array di rivelatori di particelle che agiscono su un'area molto ampia. Le

osservazioni indirette consentono quindi di osservare anche i raggi cosmici di origine extragalattica.

Da decenni, i raggi cosmici ad alta energia vengono rilevati regolarmente grazie a queste tecniche, ma il loro studio rimane complicato, soprattutto per quanto riguarda la direzione di provenienza. Infatti, essendo che i raggi cosmici di origine extragalattica vengono rilevati tramite le EAS, le loro proprietà possono essere ricavate solo indirettamente a partire dalle caratteristiche delle particelle secondarie da essi originate. Inoltre, a causa delle deviazioni dovute ai campi magnetici, la loro direzione di arrivo non può essere determinata in modo certo.

1.3 Simulare i raggi cosmici

Nell'astrofisica moderna, i metodi di osservazione appena illustrati sono affiancati da simulazioni che riproducono il verificarsi di fenomeni astronomici su larga scala, altrimenti impossibili da replicare in un laboratorio. Ciò facilità l'applicazione di un'approccio sperimentale allo studio dei raggi cosmici, consentendo un'analisi più approfondita dei processi che coinvolgono queste particelle.

Le simulazioni utilizzate in tale ambito hanno lo scopo di modellare il comportamento dei cluster di galassie: oggetti di grandi dimensioni aventi una massa pari a 10^{14} o 10^{15} volte quella del Sole e costituiti da un insieme di galassie legate tra loro dall'attrazione gravitazionale. L'evoluzione nel tempo dei cluster di galassie viene modellata sfruttando i processi di collasso gravitazionale, shock acceleration ed emissione di radiazione di sincrotrone descritti in precedenza. L'utilizzo delle simulazioni consente quindi di ricreare computazionalmente le condizioni che determinano la produzione e l'emissione di raggi cosmici all'interno di questi enormi oggetti astronomici. A partire dai cluster simulati, è poi possibile generare osservazioni ottiche, radio, a raggi X o a raggi gamma, che possono essere confrontate con le osservazioni astronomiche reali per verificare la correttezza dei modelli teorici

e quantificare i parametri in essi utilizzati. La Figura 1.2 mostra i risultati che si possono ottenere da una di queste simulazioni.

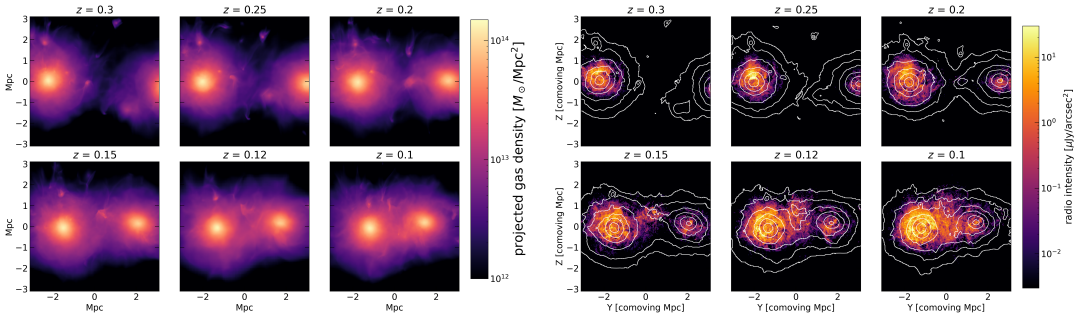

Figura 1.2: Risultati di una simulazione idrodinamica di cluster di galassie. A sinistra, 6 proiezioni che mostrano la densità dei gas all'interno delle galassie simulate. A destra, le corrispondenti osservazioni radio. Fonte: Nishiwaki et al. (in prep.).

La modellazione dei cluster di galassie si basa su diversi tipi di schemi numerici, a seconda del tipo di interazioni considerate. In particolare, si hanno due tipi di modelli: quelli che si concentrano sulla forza gravitazionale e quelli che considerano le forze idrodinamiche. Per quanto riguarda le simulazioni idrodinamiche, esse si distinguono a loro volta in due tipi di approcci [2]:

- I metodi senza griglia, o Lagrangiani, che modellano i cluster sfruttando il movimento nello spazio di particelle Lagrangiane, dette anche traccianti.

- I metodi a griglia, o Euleriani, che suddividono lo spazio in celle all'interno delle quali viene calcolata l'evoluzione degli elementi del cluster.

I metodi Lagrangiani hanno il vantaggio di non essere legati ad una rappresentazione a griglia, e quindi consentono di ottenere una risoluzione spaziale molto più elevata rispetto ai metodi Euleriani. Inoltre, a differenza, dei metodi a griglia, i metodi Lagrangiani consentono di tracciare il movimento di ogni singola particella nella simulazione. D'altra parte, in genere questi metodi comportano una qualità più bassa dal punto di vista dei fenomeni fisici simulati, mentre quelli Euleriani risultano più accurati. Solitamente,

quindi, i metodi Euleriani sono utilizzati in simulazioni su scala più ampia che richiedono un minor livello di dettaglio, ma una maggiore accuratezza a livello fisico, mentre quelli Lagrangiani sono utilizzati nelle situazioni in cui è richiesta una risoluzione spaziale elevata, e vengono applicati a casi più ristretti come cluster di piccole dimensioni o porzioni di essi. Più recentemente è stata introdotta la tecnologia dell'Adaptive Mesh Refinement (AMR), che consente di aumentare il livello di dettaglio dei metodi Euleriani senza comportare un maggior costo computazionale. L'AMR si basa infatti su celle a dimensione variabile, che consentono di avere una griglia a grana più fine nelle aree della simulazione con maggiore densità, in cui è richiesta una risoluzione più alta, e a grana più grossa nelle aree a minore densità. Questa tecnica consente quindi di risparmiare risorse computazionali, in quanto effettua un campionamento dettagliato solo delle parti più "interessanti" della simulazione. Tuttavia, l'AMR presenta alcuni svantaggi legati al fatto che la gestione della griglia adattiva è complessa, e le simulazioni presentano delle inaccuratezze nelle interfacce tra porzioni di griglia a diversa risoluzione. Inoltre, le simulazioni AMR sono difficili da parallelizzare, e di conseguenza soffrono di una scarsa scalabilità. Nonostante abbia portato diversi vantaggi all'approccio a griglia, quindi, l'introduzione dell'AMR non ha sostituito l'utilizzo dei metodi Lagrangiani, che in determinati casi sono preferibili rispetto a quelli Euleriani.

Per consentire lo studio dei raggi cosmici, nei metodi di simulazione illustrati è necessario integrare dei modelli che riproducono l'emissione di queste particelle da parte dei cluster di galassie. Rappresentare l'evoluzione dei cluster e l'emissione di raggi cosmici in un unico modello, tuttavia, è complesso, e molte simulazioni non lo supportano. In questi casi, quindi, si divide il processo di simulazione in due fasi, che per i metodi Lagrangiani sono:

1. Una prima simulazione che calcola l'evoluzione dei cluster nel tempo, in base alle caratteristiche fisiche di ogni tracciante.

2. Un'operazione di post-processing che, per ogni istante di tempo della simulazione, calcola la quantità di raggi cosmici emessi da ogni

tracciante, generando un'osservazione simulata.

In questa tesi, ci si focalizza sulla seconda fase, implementata dal codice di riferimento considerato.

1.3.1 L'equazione di Fokker-Planck per il calcolo dello spettro

L'emissione di raggi cosmici e radiazioni di sincrotrone da parte delle particelle traccianti può essere descritto come un processo di diffusione attraverso l'equazione di Fokker-Planck [7]. La risoluzione di questa equazione ci consente di calcolare lo spettro di emissione delle particelle traccianti della simulazione, generando dei dati osservabili che, se confrontati con le effettive osservazioni astronomiche, consentono di stimare con più accuratezza i parametri legati al fenomeno diffusivo. In questo modo, è possibile ottenere una maggior comprensione della natura dei processi di accelerazione dei raggi cosmici.

Per evitare ambiguità, nel resto di questa tesi verrà usato il termine "raggio cosmico" per fare riferimento ad una particella emessa durante il processo di diffusione e il termine "particella" per riferirsi ad un tracciante nella simulazione idrodinamica.

L'equazione è espressa in funzione di due variabili: il tempo t e la variabile x, che può rappresentare l'energia oppure la quantità di moto. L'equazione per il calcolo dello spettro delle particelle è la seguente:

$$\frac{\mathrm{d}u}{\mathrm{d}t} = \frac{1}{A(x)}\frac{\mathrm{d}}{\mathrm{d}x}\left(C(x)\frac{\mathrm{d}u}{\mathrm{d}x} + B(x)u\right) - \frac{u}{T(x)} + Q(x) \tag{1.1}$$

dove u corrisponde alla funzione $u(x,t)$, e $u(x,t)A(x)\,\mathrm{d}x$ descrive il numero di raggi cosmici emessi nell'intervallo $[x, x+\mathrm{d}x]$ al tempo t. $A(x)$ rappresenta il fattore di fase, e il suo valore dipende dal significato della variabile x. In particolare, $A(x)$ è pari a 1 se x rappresenta l'energia, e $4\pi x^2$ se x rappre-

senta la quantità di moto. $B(x)$, $C(x)$, $T(x)$ e $Q(x)$, invece, sono coefficienti determinati dalle condizioni fisiche di ciascuna particella. In particolare:

- $C(x)$ è il coefficiente di diffusione.
- $B(x)$ è il coefficiente di avvezione, che esprime la tendenza dei raggi cosmici a subire una deriva verso l'alto o verso il basso.
- $T(x)$ è il tasso di fuga dei raggi cosmici, che descrive il tempo necessario affinché un raggio cosmico con una certa energia venga rilasciato dalla sorgente.
- $Q(x)$ è il tasso di iniezione dei raggi cosmici, che descrive il tempo impiegato da ioni ed elettroni per raggiungere un'energia sufficiente a diventare raggi cosmici e partecipare nel processo di diffusione.

Nella formulazione di Fokker-Planck mostrata nell'equazione 1.1, questi coefficienti non dipendono da t, ma solo da x, in quanto si assume che il tempo in cui variano sia maggiore dell'intervallo di tempo considerato dal modello. Inoltre, con x nell'intervallo $[0, \infty[$, si assume che i coefficienti abbiano le seguenti caratteristiche:

- $A(x)$ e $C(x)$ devono essere > 0.
- $T(x)$ e $Q(x)$ devono essere ≥ 0.
- $B(x)$ può avere un valore qualsiasi nell'intervallo $]-\infty, \infty[$.

Il termine:

$$C(x)\frac{\mathrm{d}u}{\mathrm{d}x} + B(x)u \tag{1.2}$$

rappresenta il flusso di raggi cosmici, e si indica con $F(x,t)$.

L'equazione 1.1 presenta delle singolarità per $x = 0$ e $x = \infty$, e ciò rende problematico valutarla numericamente su tali valori di x. Per questo motivo, la soluzione dell'equazione viene svolta in un intervallo $[x_0, x_M]$, dove $0 < x_0 < x_M < \infty$, e sugli estremi x_0 e x_M si impongono delle opportune

condizioni. In particolare, una condizione che fornisce una buona approssimazione dei fenomeni fisici descritti dall'equazione è quella per cui il flusso dev'essere nullo in questi due valori, cioè:

$$F(x_0, t) = F(x_M, t) = 0 \tag{1.3}$$

Il vantaggio di questa condizione è il fatto che, rispetto alla rappresentazione che considera l'intervallo $[0, \infty[$, mantiene inalterato il numero complessivo di raggi cosmici emessi al tempo t, espresso come:

$$N(t) = \int_0^\infty A(x)\,\mathrm{d}x\,u(x,t) \tag{1.4}$$

1.3.2 Il metodo di Chang-Cooper

Il metodo di Chang-Cooper è un metodo numerico di risoluzione dell'equazione di Fokker-Planck che appartiene alla classe degli schemi a differenze finite [3]. Per poter risolvere numericamente l'equazione di Fokker-Planck, i metodi di questa classe operano una discretizzazione dei valori di t e x. In particolare, l'intervallo di tempo è diviso in vari timestep, dove il timestep con indice n è indicato come t_n, mentre l'intervallo $[x_0, x_M]$ è diviso in $M+1$ punti indicati come x_m, dove l'indice m va da 0 a M. A ciascun punto x_m corrisponde un intervallo di energia o quantità di moto, che prende il nome di "bin". Questa classe di metodi prevede di risolvere l'equazione per ogni singolo timestep, calcolando la soluzione al timestep t_{n+1} a partire da quella calcolata per t_n. La risoluzione ad un certo timestep consiste nel calcolare il numero di raggi cosmici emessi per ogni bin.

L'intervallo di tempo tra due timestep è indicato come:

$$\Delta t = t_{n+1} - t_n \tag{1.5}$$

e non è costante al variare di n, quindi dev'essere calcolato per ogni coppia di timestep.

Il punto medio tra due punti su x è individuato dalla semplice media aritmetica:

$$x_{m+1/2} = (x_{m+1} + x_m)/2 \tag{1.6}$$

La definizione del punto medio ci consente di definire l'ampiezza del bin corrispondente ad ogni punto x_m. Tale ampiezza è espressa come:

$$\Delta x_m = (x_{m+1} - x_{m-1})/2 \tag{1.7}$$

Per i coefficienti che dipendono solo da x, si utilizza la notazione:

$$A_m = A(x_m) \tag{1.8}$$

mentre per i termini che dipendono sia da x che da t, si scrive:

$$u_m^n = u(x_m, t_n) \tag{1.9}$$

In base a questa notazione, l'equazione di Fokker-Planck può essere discretizzata nel modo seguente:

$$\frac{u_m^{n+1} - u_m^n}{\Delta t} = \frac{1}{A_m} \frac{F_{m+1/2}^{n+1} - F_{m-1/2}^{n+1}}{\Delta x_m} - \frac{u_m^{n+1}}{T_m} + Q_m \tag{1.10}$$

per ogni $m = 0, \ldots, M$, dove $F_{m+1/2}^{n+1} = (F_m^{n+1} + F_{m+1}^{n+1})/2$.

I diversi schemi a differenze finite si distinguono in base al modo in cui definiscono il flusso al timestep successivo, cioè F_m^{n+1}. In ognuno di essi, sostituendo il flusso nell'equazione di Fokker-Planck discretizzata, si ottiene un sistema tridiagonale di equazioni lineari, che può essere scritto come:

$$\begin{cases} -a_m u_{m-1}^{n+1} + b_m {u_m}^n + 1 - c_m u_{m+1}^{n+1} = r_m \\ a_0 = c_M = 0 \end{cases} \tag{1.11}$$

dove r_m è una funzione di u_m^n. Questo sistema può essere risolto tramite un algoritmo di eliminazione Gaussiana con sostituzione all'indietro, che per i

sistemi tridiagonali presenta una variante efficiente avente una complessità di $O(M)$ invece che $O(M^3)$ [8].

In particolare, il metodo di Chang-Cooper definisce una discretizzazione del flusso che considera per il termine $C(x)$ una differenza centrata tra u_{m+1}^{n+1} e u_m^{n+1}, e per il termine $B(x)$ una differenza pesata in base ad un termine δ_m. L'equazione del flusso è quindi la seguente:

$$F_{m+1/2}^{n+1} = B_{m+1/2}[(1-\delta_{m+1/2})u_{m+1}^{n+1} + \delta_{m+1/2}u_m^{n+1}] + C_{m+1/2}\frac{u_{m+1}^{n+1} - u_m^{n+1}}{\Delta x_{m+1/2}} \tag{1.12}$$

dove $\Delta x_{m+1/2} = x_{m+1} - x_m$, e $\delta_{m+1/2}$ è definito come:

$$\delta_{m+1/2} = \frac{1}{w_{m+1/2}} - \frac{1}{exp(w_{m+1/2}) - 1} \tag{1.13}$$

con:

$$w_{m+1/2} = \frac{B_{m+1/2}}{C_{m+1/2}}\Delta x_{m+1/2} \tag{1.14}$$

Questa formulazione, con qualche aggiustamento per adattarla all'aritmetica a virgola mobile, fornisce un metodo accurato e numericamente stabile per il calcolo dello spettro, consentendone l'utilizzo in applicazioni pratiche che richiedono un'elevata precisione nella risoluzione di Fokker-Planck su molti ordini di grandezza di x [7].

Capitolo 2

Codice di riferimento

Il codice considerato in questa tesi si inserisce nel contesto di una simulazione idrodinamica di cluster di galassie basata su un metodo Lagrangiano. In particolare, l'obiettivo del codice è implementare l'operazione di post-processing che consente di calcolare la quantità di raggi cosmici e radiazioni emesse da ogni particella della simulazione. Questa operazione è necessaria in quanto la simulazione idrodinamica non supporta le computazioni con i raggi cosmici. Tale simulazione calcola l'evoluzione delle galassie su diversi timestep, e per ciascuno di essi restituisce in output le caratteristiche fisiche di ogni particella. Per ciascuna particella, il codice di riferimento calcola lo spettro di emissione risolvendo l'equazione di Fokker-Planck su tutti i timestep della simulazione. La risoluzione di Fokker-Planck è svolta sfruttando il metodo di Chang-Cooper, e considerando x come la quantità di moto. Per questo motivo, i bin su x prendono il nome di "momentum bin".

2.1 Caratteristiche

Il codice è scritto in linguaggio C, e si compone di diversi moduli che implementano le funzioni matematiche per il calcolo dello spettro delle particelle.

2.1.1 Input

I dati di input derivati dalla simulazione iniziale si trovano in formato HDF5, che prevede un'organizzazione gerarchica dei dati ed è particolarmente adatto per memorizzare dataset di grandi dimensioni. Ogni file di input contiene le informazioni sullo stato della simulazione in uno specifico istante di tempo. In particolare, in ogni file sono memorizzati una serie di array monodimensionali, ognuno dei quali contiene i valori di una determinata grandezza fisica per tutte le particelle della simulazione. Alcune delle grandezze fisiche considerate sono temperatura, densità di massa e velocità. Oltre ai file contenenti i dati da elaborare, l'input del programma si compone di un file di configurazione in formato testuale, che consente di gestire diversi aspetti legati all'esecuzione del codice. I parametri presenti in tale file comprendono:

- Il percorso in cui si trovano i file di input.
- Il percorso in cui memorizzare i file di output.
- Il numero di particelle nella simulazione.
- Gli istanti di tempo (o timestep) iniziali e finali della simulazione.
- Una serie di parametri che configurano il modello matematico.
- Una serie di parametri legati al parallelismo, tra cui il numero di thread OpenMP.

2.1.2 Output

L'output del programma è costituito da tre array tridimensionali che contengono i dati relativi agli spettri di emissione delle particelle per quanto riguarda elettroni, protoni e radiazione di sincrotrone. Le tre dimensioni degli array sono:

- Indice della particella.

- Istante di tempo.

- Momentum bin.

Quindi, in ogni elemento di uno di questi array è memorizzata la quantità di raggi cosmici o radiazioni emesse da una data particella, in un dato istante di tempo e ad un certo momentum bin. Il numero di bin, corrispondente al parametro M del metodo di Chang-Cooper, è costante, e pari a 128, mentre il numero di particelle e istanti di tempo varia a seconda della simulazione. I dati contenuti nei tre array di output vengono salvati in altrettanti file in formato binario. Per ottimizzare i tempi di elaborazione e scrittura, gli array di output vengono in realtà gestiti come array bidimensionali, ricavati unendo due degli assi originali in uno solo. In questa rappresentazione, l'asse delle righe corrisponde all'indice della particella, mentre l'asse delle colonne è indicizzato moltiplicando l'istante di tempo per il bin. Di conseguenza, l'array finale contiene una riga per ogni particella, e in ogni riga sono memorizzati gruppi di 128 valori, ognuno contenente le informazioni su tutti i bin in un dato istante di tempo. Quindi, le prime 128 celle corrispondono allo spettro nell'istante iniziale, le seguenti 128 contengono lo spettro nell'istante successivo, e così via. In Figura 2.1 è mostrata una rappresentazione schematica della struttura degli array di output.

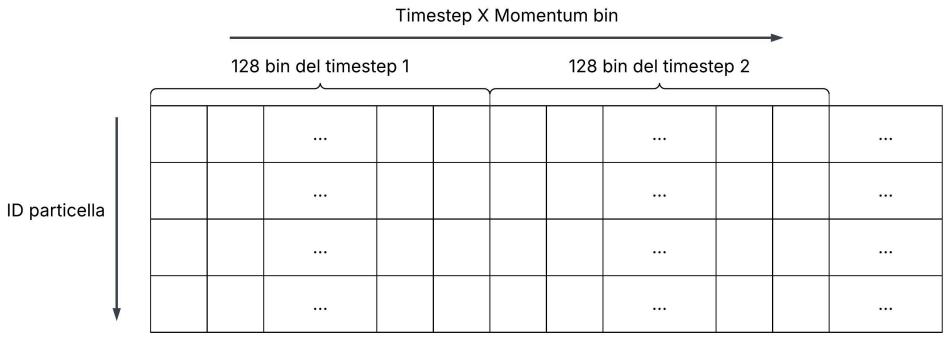

Figura 2.1: Struttura degli array di output.

2.1.3 Algoritmo e complessità

Per agevolare le spiegazioni riportate nei paragrafi seguenti, introduciamo la seguente notazione:

N numero di particelle nella simulazione.

t_i istante di tempo iniziale della simulazione.

t_f istante di tempo finale della simulazione.

N_t numero di timestep nella simulazione. I timestep considerati sono una discretizzazione dell'intervallo $[t_i, t_f]$.

In fase di lettura, i dati presenti nei file di input vengono memorizzati in una serie di array bidimensionali, uno per ogni grandezza fisica. Ogni array contiene un numero di righe pari al numero di timestep nella simulazione e un numero di colonne pari al numero di particelle. Di conseguenza, la computazione effettuata dal programma viene svolta su una serie di matrici bidimensionali, la cui dimensione è pari a $N_t \times N$. La Figura 2.2 mostra la struttura degli array appena descritti.

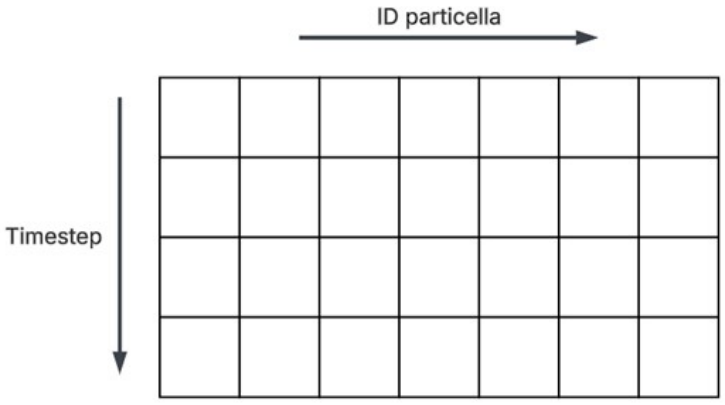

Figura 2.2: Struttura degli array bidimensionali utilizzati durante le computazioni.

La struttura generale dell'algoritmo per il calcolo dello spettro di emissione delle particelle è la seguente:

for $i := 0$ to $N - 1$ **do**
    Inizializza lo spettro iniziale per la particella i
    Normalizza lo spettro iniziale per i
    for $j := 0$ to $N_t - 1$ **do**
        Imposta i coefficienti di Chang-Cooper per particella i e timestep j
        Applica Chang-Cooper per i e j

Il ciclo esterno itera su tutte le particelle, mentre il ciclo interno calcola lo spettro di emissione di una specifica particella per ogni timestep. Il codice deve quindi risolvere un'equazione di Fokker-Planck per ogni iterazione del ciclo interno. Le computazioni svolte nel corpo di tale ciclo, che applicano il metodo di Chang-Cooper risolvendo il relativo sistema tridiagonale mediante eliminazione Gaussiana, sono quelle più onerose per quanto riguarda il carico di lavoro. Tuttavia, sia le fasi di inizializzazione e normalizzazione dello spettro iniziale che quelle di risoluzione dell'equazione di Fokker-Planck hanno una complessità espressa in funzione di M, che essendo costante non dipende dalla dimensione del problema. La complessità asintotica dell'algoritmo corrisponde quindi a $\Theta(N \times N_t)$. Ciò è dovuto al fatto che per ognuna delle N particelle è necessario eseguire N_t volte il calcolo dello spettro in uno specifico timestep. Le operazioni svolte all'interno dei cicli non pesano sulla complessità asintotica perché la loro complessità è costante.

2.1.4 Parallelismo

Nel codice di riferimento sono implementati due tipi di parallelismo CPU, grazie all'utilizzo di apposite librerie per il calcolo parallelo. Infatti, il codice sfrutta sia il parallelismo a memoria condivisa offerta da OpenMP, sia il parallelismo a memoria distribuita di MPI.

Il parallelismo di OpenMP prevede che le unità di esecuzione siano costituite da thread che fanno parte di un unico processo e che condividono un unico spazio di memoria in cui immagazzinare i dati. La condivisione della memoria consente ai vari thread di comunicare tra loro sfruttando delle variabili globali, e quindi lo scambio di informazioni avviene in modo rapido e relativamente semplice.

MPI, invece, è utilizzato su architetture a memoria distribuita, in cui le unità di esecuzione sono spesso costituite da dispositivi connessi per mezzo di una rete. In MPI, l'esecuzione del codice avviene su più processi diversi, ognuno avente un proprio spazio di memoria separato rispetto agli altri. L'assenza di uno spazio di memoria condiviso impedisce l'utilizzo di variabili globali, e comporta la necessità di gestire la comunicazione tra processi mediante lo scambio esplicito di messaggi. Ciò rende le operazioni di comunicazione più costose in termini di prestazioni, ma consente di sfruttare contemporaneamente la potenza di calcolo di più dispositivi.

Attraverso i parametri del file di configurazione, è possibile specificare quale tipo di parallelismo utilizzare, e si ha anche la possibilità di usarli in combinazione per ottenere un ulteriore speedup.

La strategia di suddivisione del carico di lavoro utilizzata dai due tipi di parallelismo è molto simile: sia nel caso MPI che nel caso OpenMP, le particelle da elaborare sono equamente distribuite tra le varie unità di esecuzione disponibili. Ogni unità di esecuzione, dunque, svolge le computazioni sul proprio sottoinsieme di particelle. Questa suddivisione è resa possibile dal fatto che il calcolo dello spettro sulle diverse particelle è embarassingly parallel, e ciò vuol dire che ogni particella può essere elaborata in modo indipendente rispetto alle altre. La differenza tra i due tipi di parallelismo

implementati nel codice di riferimento risiede, quindi, unicamente nel paradigma di programmazione parallela adottato. Applicando in combinazione OpenMP ed MPI, le particelle vengono suddivise dapprima tra i vari nodi MPI, e successivamente tra i vari thread all'interno di ogni nodo.

2.2 Test iniziali

Per verificarne il corretto funzionamento, il codice di riferimento è stato testato su due architetture differenti. In particolare, i test e la successiva valutazione delle prestazioni sono stati effettuati sia su un PC convenzionale, sia sul supercomputer Leonardo, messo a disposizione dal CINECA. In Tabella 2.1 sono riportate le caratteristiche hardware di entrambe le macchine. Le specifiche di Leonardo sono relative ad un singolo nodo della partizione Booster, dotata di GPU. Le GPU usate da Leonardo utilizzano un chip basato sull'architettura Ampere A100 di Nvidia e leggermente modificato per ottenere prestazioni migliori rispetto ad un A100 convenzionale [9].

	PC	**Leonardo**
Processore	AMD Ryzen 7 3700X	Intel Ice Lake Xeon Platinum 8358
Frequenza di clock CPU	3.59 GHz	2.60 GHz
Numero di core fisici	8	32
GPU	Nvidia GTX 1660 Super	4 x Nvidia Ampere A100 custom

Tabella 2.1: Caratteristiche dell'hardware utilizzato

Per verificare la correttezza dei risultati sia in questa fase che in fase di test del codice GPU, è stato realizzato un codice in linguaggio C che confronta l'output di riferimento con l'output restituito dal programma. Il confronto viene effettuato prendendo in considerazione una certa soglia di tolleranza: se la differenza tra due valori è maggiore della tolleranza specificata, il codice segnala l'errore in un file di log. L'utilizzo di questa soglia di tolleranza

consente di limitare il numero di falsi positivi dovuti agli errori di approssimazione dell'aritmetica discreta, che si possono verificare se si modifica l'ordine delle operazioni oppure si effettuano test su architetture diverse. Per visualizzare i risultati, è stato inoltre realizzato un codice in linguaggio Python con l'obiettivo di mostrare in un grafico lo spettro di emissione di una o più particelle in un determinato istante di tempo. I grafici sono stati realizzati sfruttando il modulo matplotlib.pyplot. Il dataset di test, su cui sono state anche valutate le prestazioni del programma CPU, comprende 5233 particelle, $t_i = 13$ e $t_f = 113$. Di conseguenza, la computazione viene svolta su $N_t = 100$ timestep. Sebbene questo dataset abbia dimensioni ridotte rispetto ai casi reali, il tempo di esecuzione del codice seriale su tutti i timestep risulta piuttosto elevato (da 3 a 10 ore, a seconda dell'architettura). Per questo motivo, i primi test di verifica sono stati effettuati limitando il numero di timestep, in modo che su ogni particella venissero svolte poche iterazioni. Il codice è stato testato prima sul PC, e successivamente su Leonardo. Dopo aver verificato la correttezza sul caso ridotto, si è passati all'esecuzione del codice sull'intero dataset, quindi impostando il timestep finale a 113. Eseguendo il codice di visualizzazione dati sull'output, si ottengono i grafici riportati in Figura 2.3, nei quali è mostrato lo spettro di emissione relativo agli elettroni. I grafici sono rappresentati in scala logaritmica, e mostrano come lo spettro descriva una legge di potenza, cioè una linea retta, che va poi a decadere nei bin più alti. Questo è il risultato che ci aspettiamo di osservare da questo tipo di simulazione.

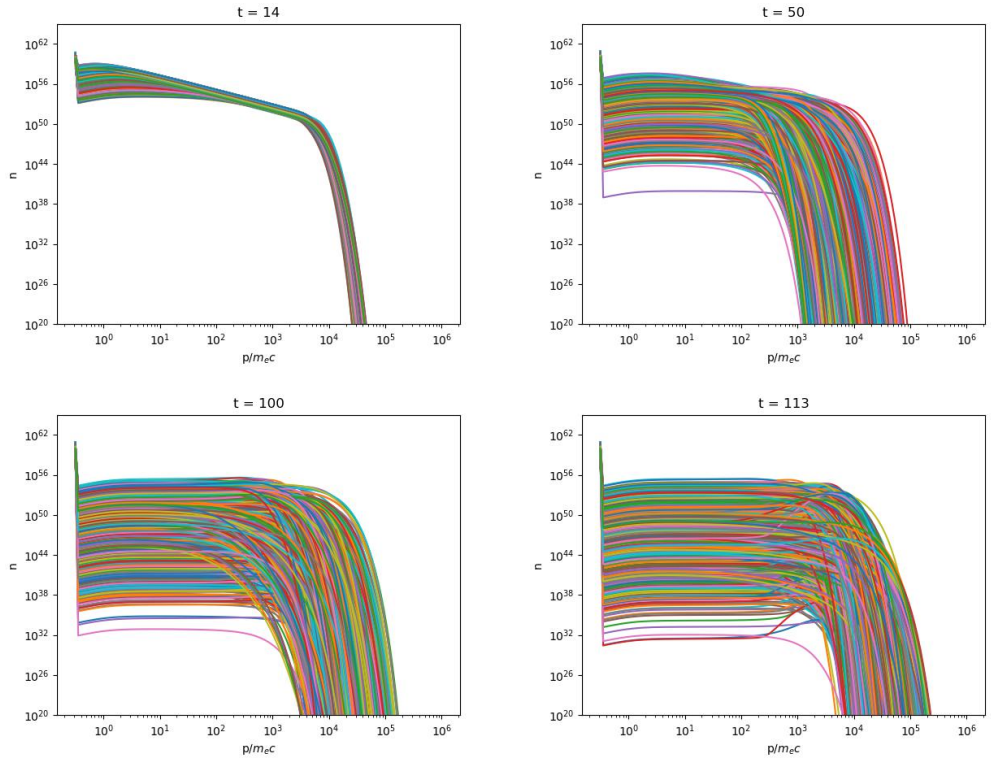

Figura 2.3: Grafici dello spettro di emissione di elettroni per tutte le particelle del dataset in diversi istanti di tempo t (14, 50, 100 e 113). Nel grafico, n è il numero di elettroni emessi, mentre $p/m_e c$ rappresenta il momentum bin. Ogni linea sul grafico descrive lo spettro di una singola particella.

2.3 Valutazione delle prestazioni del codice CPU

La valutazione delle prestazioni del codice di riferimento ha due obiettivi principali:

- Valutare le prestazioni del parallelismo OpenMP, in modo da poterlo successivamente confrontare con il parallelismo GPU.
- Confrontare l'efficienza dei due tipi di parallelismo già implementati nel codice.

Come già detto, il parallelismo OpenMP e quello MPI operano la stessa suddivisione del carico di lavoro, e ciò consente un confronto accurato tra due diversi approcci di programmazione parallela. Le prestazioni del codice sono state misurate in termini di speedup, strong scaling efficiency e weak scaling efficiency.

Lo speedup è una misura che fornisce un criterio di confronto tra un'implementazione seriale e una parallela. In particolare, lo speedup con p processori, indicato come $S(p)$, si definisce nel modo seguente:

$$S(p) = \frac{T(1)}{T(p)} \tag{2.1}$$

dove p è il numero di processori, $T(1)$ è il tempo di esecuzione del programma parallelo eseguito con un processore e $T(p)$ è il tempo di esecuzione del programma parallelo eseguito con p processori.

La strong scaling efficiency misura l'efficienza dell'implementazione parallela all'aumentare del numero di processori utilizzati per risolvere lo stesso problema. Il calcolo della strong scaling efficiency richiede quindi di mantenere fissa la dimensione del problema e aumentare progressivamente il numero di unità di esecuzione. Il valore dell'efficienza per ogni misura effettuata viene calcolato come:

$$E(p) = \frac{S(p)}{p} \tag{2.2}$$

La weak scaling efficiency, invece, misura l'efficienza del programma parallelo mantenendo costante il carico di lavoro svolto da ciascuna unità di esecuzione. Per misurarla, dunque, è necessario definire un'unità di lavoro, cioè la quantità di lavoro che dev'essere assegnata ad ogni processore, ed effettuare le misure aumentando la dimensione del problema in modo proporzionale al numero di unità di esecuzione. Il valore della weak scaling efficiency viene poi ricavato nel modo seguente:

$$W(p) = \frac{T_1}{T_p} \tag{2.3}$$

Dove T_1 è il tempo impiegato da un processore per eseguire una singola unità di lavoro, e T_p è il tempo impiegato da p processori per eseguire p unità di lavoro. I tempi di esecuzione sono stati misurati sul dataset di test già citato in precedenza.

2.3.1 Speedup e strong scaling efficiency

Il calcolo di speedup e strong scaling efficiency è stato effettuato su due diversi casi di test: il primo avente timestep finale pari a 14, quindi $N_t = 1$, e il secondo con timestep finale pari a 113, che dunque considera il dataset completo. Entrambi i casi di test sono stati eseguiti su tutte le 5233 particelle.

Per il primo caso di test, su entrambe le macchine riportate in Tabella 2.1, sono state valutate sia le prestazioni di OpenMP, sia quelle di MPI. Per poter svolgere la valutazione di MPI, l'output del codice è stato leggermente modificato in modo da stampare il tempo di esecuzione medio tra i vari nodi. Su Leonardo, le prestazioni di MPI sono state misurate aumentando il numero di nodi fisici del cluster e allocando su ognuno di essi un solo core fisico (quindi mantenendo OMP_NUM_THREADS = 1). Sul PC, invece, si dispone di una sola CPU, quindi i nodi MPI corrispondono ai diversi core fisici al suo interno. La Figura 2.4 riporta l'andamento di tempo di esecuzione, speedup e strong scaling efficiency nel primo caso di test.

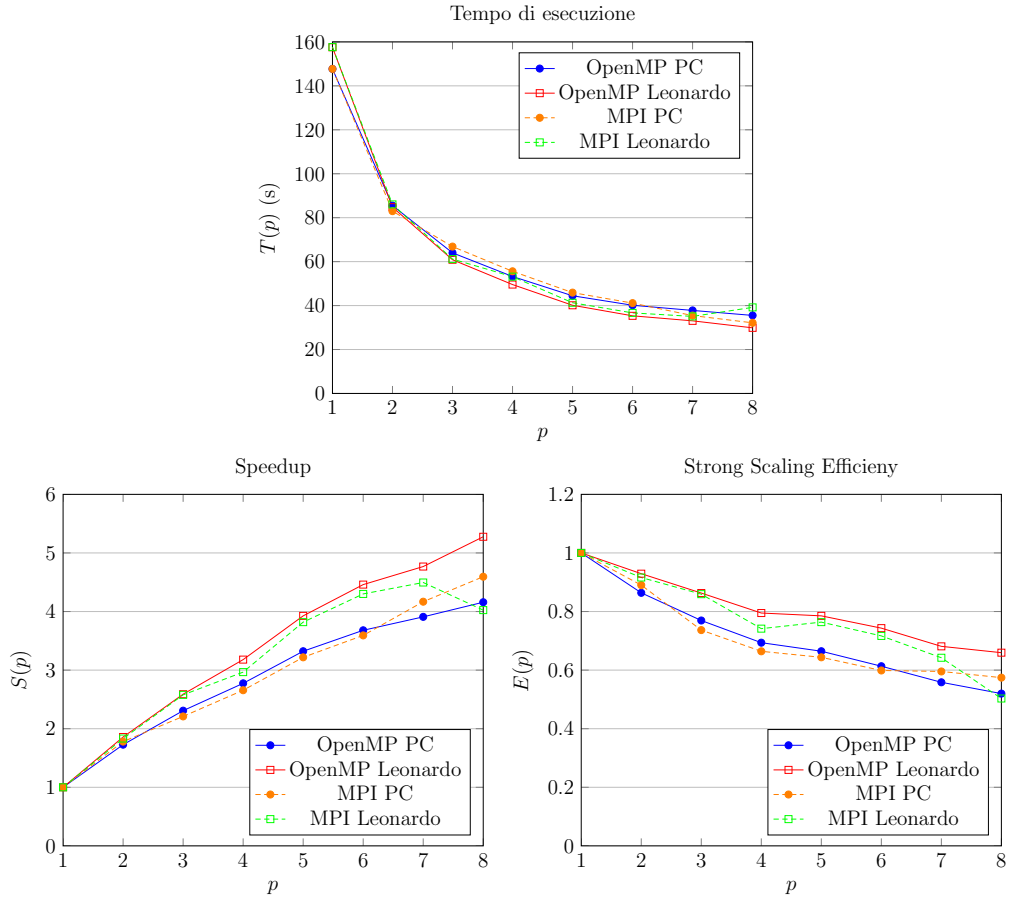

Figura 2.4: Tempo di esecuzione, speedup e strong scaling efficiency con $t_f = 14$. Nei grafici, p fa riferimento al numero di unità di esecuzione (core per OpenMP, nodi per MPI), mentre $T(p)$, $S(p)$ e $E(p)$ fanno riferimento rispettivamente a tempo di esecuzione, speedup e strong scaling efficiency con p unità. Il tempo di esecuzione è espresso in secondi (s).

Per il secondo caso di test, che costituisce il dataset completo, le prestazioni sono state valutate unicamente su Leonardo. In questo modo, è stato possibile effettuare i test in un tempo ragionevole, sfruttando un grado di parallelismo più elevato rispetto a quello offerto dalla CPU del PC. Anche in questo caso sono state misurate sia le prestazioni di OpenMP che quelle di MPI. Anche in questo caso sono state misurate sia le prestazioni di OpenMP che quelle di MPI. Dato il numero relativamente elevato di core disponibili in un nodo di Leonardo, si è scelto di aumentare le unità di esecuzione in modo

esponenziale, via via raddoppiando il valore di p, fino ad un massimo di 32. Per la valutazione con MPI, i test con 16 e 32 nodi sono stati effettuati su 8 nodi fisici lanciando più processi su ogni nodo. La scelta di limitare il numero di nodi fisici a 8 è stata fatta per evitare di allocare una quantità eccessiva di risorse del cluster. La Figura 2.5 mostra i risultati dei test eseguiti sul dataset completo.

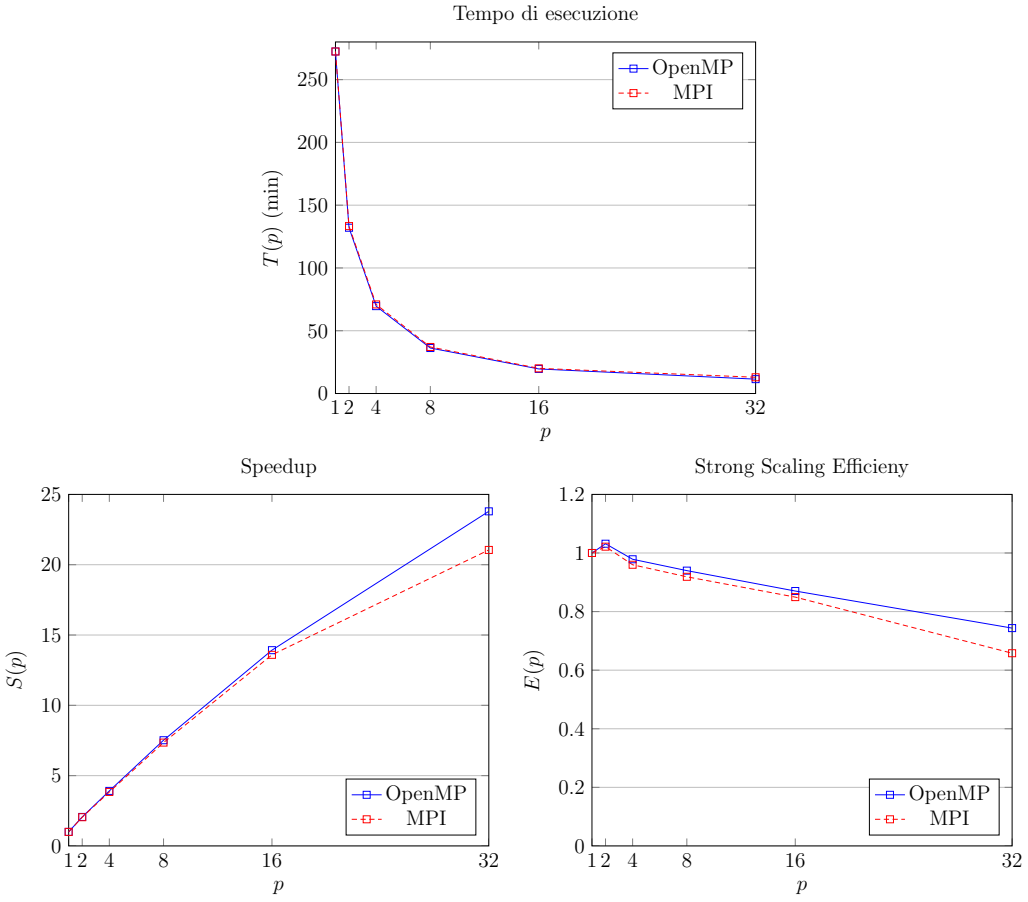

Figura 2.5: Tempo di esecuzione, speedup e strong scaling efficiency del codice eseguito sul dataset completo su Leonardo. In questo caso, il tempo di esecuzione è espresso in minuti (min).

2.3.2 Weak scaling efficiency

Il calcolo della weak scaling efficiency richiede di aumentare il carico di lavoro dell'algoritmo in modo proporzionale al numero di unità di esecuzione. Come già spiegato in precedenza, il costo computazionale dell'algoritmo può essere definito come $\Theta(N \times N_t)$; dunque, il carico di lavoro può essere modificato sia aumentando il numero di particelle considerate, sia aumentando il numero di timestep su cui vengono svolte le computazioni. A causa delle caratteristiche del codice, l'approccio più immediato è quello di mantenere invariato il numero di particelle e aumentare il numero di timestep in modo proporzionale. L'unità di lavoro scelta per effettuare i test corrisponde a tre timestep per unità di esecuzione. Quindi, ad ogni unità di esecuzione è assegnato un carico di lavoro costante pari a $N \times 3$. La valutazione della weak scaling efficiency è stata effettuata su Leonardo, confrontando le prestazioni di OpenMP ed MPI. Inoltre, come nel caso precedente, le misure con 16 e 32 nodi MPI sono state effettuate allocando 8 nodi fisici con più processi per ogni nodo. La Figura 2.6 mostra l'andamento della weak scaling efficiency per entrambi i tipi di parallelismo.

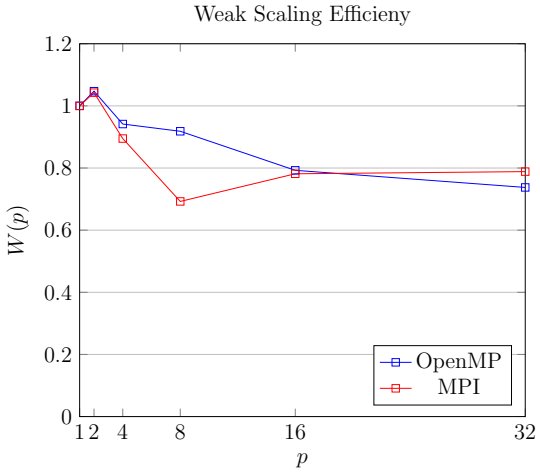

Figura 2.6: Weak scaling efficiency del programma eseguito con OpenMP ed MPI. Nel grafico, $W(p)$ fa riferimento alla weak scaling efficiency con p unità di esecuzione.

2.3.3 Considerazioni

Dai risultati ottenuti nella fase di valutazione delle prestazioni, si evince che i due tipi di parallelismo implementati nel codice sono molto simili per quanto riguarda l'efficienza. I test sul dataset completo, con $N_t = 100$, mostrano in entrambi casi valori massimi di speedup superiori a 20. Inoltre, osservando il grafico della strong scaling efficiency, si nota che l'andamento dell'efficienza per i due tipi di parallelismo è quasi analogo, e l'unica differenza significativa si riscontra nel caso con 32 unità di esecuzione, in cui il parallelismo OpenMP risulta più efficiente rispetto a quello MPI. Questa perdita di efficienza del parallelismo MPI nel caso con 32 nodi è legata all'overhead causato dall'elevato numero di operazioni di comunicazione e sincronizzazione necessarie per implementare il parallelismo a memoria distribuita. Per quanto riguarda la weak scaling efficiency, notiamo che l'andamento dell'efficienza del parallelismo OpenMP risulta più lineare rispetto all'efficienza della versione MPI, soprattutto per valori intermedi di p (4 e 8). Tuttavia, nei test con $p = 32$, possiamo notare come la weak scaling efficiency della versione MPI sia leggermente superiore rispetto a quella della versione OpenMP. Quindi, se si mantiene costante il carico di lavoro su ogni nodo, il parallelismo MPI riesce a scalare leggermente meglio rispetto al parallelismo OpenMP. Osservando i grafici di strong scaling e weak scaling efficiency nel caso $N_t = 100$, si nota, inoltre, che per $p = 2$ l'efficienza è superiore a 1, indicando uno speedup superlineare. Molto probabilmente, ciò è dovuto al fatto che, nel caso con $p = 2$, la quantità di cache totale a disposizione raddoppia rispetto al caso seriale, riducendo il numero di accessi in memoria centrale e quindi migliorando l'efficienza. Questo effetto non è presente, tuttavia, per valori di p più alti, in quanto l'overhead causato dalle operazioni di comunicazione e sincronizzazione supera i benefici apportati da una maggior quantità di cache. Un'ulteriore considerazione che si può fare confrontando i dati della strong scaling efficiency tra i due casi di test considerati ($N_t = 1$ e $N_t = 100$) è il fatto che, soprattutto nel caso di MPI, il codice ha un'efficienza più elevata per dataset di dimensioni maggiori. Questo è un risultato che ci si aspetta,

ed è dovuto all'overhead legato alla gestione del parallelismo. Su dataset di piccole dimensioni, infatti, le operazioni di allocazione e sincronizzazione dei thread o dei nodi occupano una percentuale più alta del tempo di esecuzione rispetto ai casi con molti dati, riducendo l'efficienza del programma.

Capitolo 3

Versione GPU

Effettuata l'analisi del codice di riferimento, si è passati all'implementazione di una versione in grado di sfruttare il parallelismo massivo delle GPU. La realizzazione di tale versione è stata svolta facendo uso di una tecnologia relativamente recente: le funzionalità OpenMP per la programmazione GPU. Dunque, l'obiettivo di questa fase è stato valutare l'efficacia di OpenMP nel parallelizzare tramite GPU un codice di produzione complesso e di grandi dimensioni, come quello che si considera in questa tesi.

3.1 Architettura di una GPU

Una GPU (Graphics Processing Unit) è un dispositivo di calcolo specializzato per l'esecuzione di applicazioni grafiche e per il calcolo parallelo ad alte prestazioni. Nello specifico, le GPU moderne prendono il nome di GPGPU (General-Purpose GPU), in quanto sono dotate di un'architettura che consente di sfruttare la loro capacità computazionale per scopi che vanno oltre il solo rendering grafico. L'elevata capacità computazionale delle GPU è dovuta all'elevato grado di parallelismo in esse implementato, che le rende ideali per eseguire computazioni embarassingly parallel su una grande quantità di dati. L'architettura generale di una GPU si compone di:

- Un numero elevato di Streaming Multiprocessor (SM), unità di calcolo dotate di numerosi core e in grado di eseguire in parallelo.
- Una cache L2 condivisa tra i vari SM.
- Una memoria globale di grandi dimensioni utilizzata per immagazzinare tutti i dati necessari per svolgere le computazioni.

Ciascuno Streaming Multiprocessor dispone di un numero elevato di core, ognuno in grado di eseguire operazioni elementari su un insieme ridotto di dati. Oltre ai core, l'architettura di uno SM comprende:

- Un numero elevato di registri, utilizzati dai vari core per svolgere le computazioni.
- Una cache L1 condivisa tra i vari core, detta anche Shared Memory.

La presenza della shared memory consente di ottimizzare le operazioni di lettura e scrittura sui dati, limitando il numero di accessi alla memoria centrale, più lenta. Infatti, i core di uno stesso SM possono lavorare su dati condivisi in shared memory e trasferirli in memoria centrale solo quando necessario. La Figura 3.1 mostra uno schema semplificato dell'architettura descritta.

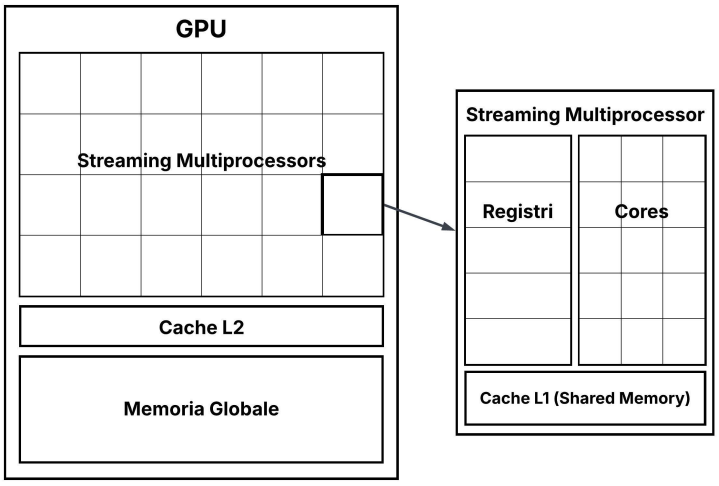

Figura 3.1: Schema semplificato dell'architettura di una GPU.

La presenza di numerosi core su ogni SM consente di ottenere un parallelismo a grana fine, basato sull'esecuzione simultanea di molte operazioni semplici. Questa caratteristica distingue il parallelismo GPU da quello CPU, basato invece su un numero ridotto di unità di esecuzione che svolgono in parallelo operazioni più complesse. Inoltre, a differenza di quello CPU, il parallelismo GPU è gerarchico, in quanto può essere gestito sia a livello degli SM, sia a livello dei singoli core all'interno di ciascun SM.

Nelle sezioni seguenti, verrà usato il termine "host" per fare riferimento alla CPU e alla sua memoria, e il termine "device" per indicare la GPU.

3.2 Programmazione GPU con OpenMP

Come accennato in Sottosezione 2.1.4, OpenMP è una libreria di C, C++ e Fortran per il calcolo parallelo ad alte prestazioni nata per consentire il parallelismo CPU secondo un paradigma a memoria condivisa. A partire dalla versione 4.0, OpenMP supporta anche il parallelismo su sistemi eterogenei, in particolare sulle GPU, e quindi ha recentemente affiancato tecnologie più consolidate quali:

- CUDA, la libreria proprietaria di Nvidia.
- ROCm, che consente di programmare su GPU AMD.
- OpenCL, un'alternativa portabile e open source.
- OpenACC, una libreria che offre un maggior livello di astrazione rispetto alle precedenti.

Allo stesso modo di OpenACC, OpenMP offre funzionalità più ad alto livello, che consentono di eseguire codice sulla GPU in modo indipendente dall'architettura dell'hardware utilizzato. L'obiettivo di OpenMP è quindi quello di fornire un maggior grado di portabilità e una maggior facilità di utilizzo rispetto a tecnologie più convenzionali.

OpenMP, come OpenACC, sfrutta un approccio basato sulle direttive. Questo vuol dire che il suo utilizzo non prevede di scrivere codice in modo esplicito, bensì di inserire nel codice esistente delle opportune direttive che comunicano al compilatore quali porzioni del programma devono essere parallelizzate. Sulla base delle direttive inserite, il compilatore si occupa poi di svolgere le opportune modifiche e ottimizzazioni per ottenere il risultato richiesto.

3.2.1 Gerarchia di parallelismo

OpenMP fornisce un parallelismo basato su diversi livelli di astrazione, che riflettono l'organizzazione gerarchica dell'architettura di una GPU. L'unità di esecuzione di base è rappresentata dal **thread**. Ogni thread esegue autonomamente una data sezione di codice, e può condividere dei dati con altri thread, attraverso l'utilizzo della shared memory. Ciascun thread OpenMP corrisponde ad uno o più core sulla GPU. I thread sono poi raggruppati in **team**, delle unità di esecuzione di livello più alto che operano in modo indipendente. Quindi, mentre i thread di uno stesso team possono sincronizzarsi e condividere gli stessi dati, più team diversi sono completamente indipendenti tra loro. Di conseguenza, non esistono meccanismi nativi di condivisione dati e sincronizzazione tra team. Questa separazione tra thread e team presenta diversi vantaggi, in quanto fornisce al programmatore un maggior controllo sul grado di parallelismo adottato, e consente anche di ottimizzare gli accessi in memoria, migliorando le prestazioni. Infatti, se più thread di uno stesso team leggono i dati dalla shared memory, si limita molto il numero di accessi alla memoria centrale, i quali rappresentano un'operazione molto onerosa in termini di prestazioni. Nel paradigma OpenMP, la gestione della shared memory a disposizione dei thread è completamente trasparente al programmatore, in quanto è svolta dal compilatore in modo automatico. Un altro vantaggio dei team è la possibilità di far eseguire più team diversi sullo stesso SM, in modo che condividano le risorse hardware. In questo modo, quando un team è in attesa di operazioni di input/output,

gli altri possono sfruttare la potenza di calcolo dei core, riducendo la latenza nelle operazioni di accesso in memoria. Oltre a team e thread, OpenMP offre un livello di parallelismo ancora più fine, rappresentato dalle **istruzioni SIMD**, che forniscono un parallelismo di tipo vettoriale. Questo meccanismo consente di eseguire la stessa istruzione su più dati contemporaneamente, aumentando ulteriormente l'efficienza. Mediante opportune direttive, OpenMP dà la possibilità di scegliere il livello di parallelismo da utilizzare in ciascun punto del codice.

3.2.2 Mapping dei dati

Un altro aspetto fondamentale da considerare quando si scrive codice per la GPU è il trasferimento dei dati dal dispositivo host al device e viceversa. Quando l'esecuzione giunge in corrispondenza di una regione parallela sulla GPU, cioè una porzione di codice che deve eseguire in parallelo, OpenMP si occupa di trasferire sul device tutti i dati necessari per la corretta esecuzione del codice. In particolare, i dati considerati da OpenMP sono quelli relativi a variabili dichiarate fuori dalla regione parallela e richiamate al suo interno. Su questi dati, OpenMP opera una distinzione in tre tipi:

Scalar: Semplici valori numerici interi o in virgola mobile.

Pointer: Puntatori ad una determinata locazione di memoria.

Aggregate: Dati più complessi, come array statici o strutture.

Il comportamento di default di OpenMP prevede che ogni dato venga copiato dall'host alla GPU all'inizio della regione parallela e poi trasferito nuovamente sull'host una volta che l'esecuzione del codice GPU è terminata. Se la quantità di dati da trasferire è elevata, questa gestione può comportare problemi di efficienza dovuti a trasferimenti superflui di dati tra host e device, specialmente se il collegamento tra i due dispositivi avviene per mezzo di un bus di comunicazione con banda limitata. In generale, infatti, non tutti i dati che vengono passati in input alla GPU necessitano di essere copiati sull'host

al termine delle computazioni. Un altro aspetto problematico del comportamento di default è legato alla gestione dei puntatori ad aree di memoria allocate dinamicamente. Infatti, mentre i dati di tipo "aggregate" vengono copiati nella loro interezza, per i puntatori viene copiato solo l'indirizzo di memoria a cui fanno riferimento. Di conseguenza, il contenuto degli array dinamici non viene copiato automaticamente sulla GPU.

Per far fronte a questi problemi, OpenMP mette a disposizione delle apposite direttive di mapping, che consentono di specificare il modo con cui i dati devono essere mappati sulla GPU. Il mapping esplicito dei dati viene effettuato mediante il costrutto `map`, che consente di specificare i dati da mappare sulla GPU e il tipo di mapping da utilizzare. I principali tipi di mapping disponibili sono:

to: Copia i dati dall'host al device.

from: Copia i dati dal device all'host.

tofrom: Copia i dati dall'host al device all'inizio della regione parallela e poi dal device all'host al termine. Questa è l'impostazione di default di OpenMP.

alloc: Alloca semplicemente la memoria sul device, senza trasferire dati.

L'utilizzo opportuno di queste politiche di mapping consente di limitare il numero di trasferimenti effettuati, aumentando l'efficienza del programma. Inoltre, le direttive di mapping consentono di specificare al compilatore la dimensione delle aree di memoria che devono essere copiate, rendendo possibile l'utilizzo di array dinamici.

3.3 Strategia di parallelismo adottata

Come mostrato nel Capitolo 2, i dati di input hanno una struttura bidimensionale, e il codice di riferimento organizza le computazioni in due cicli annidati. Apparentemente, la suddivisione del carico di lavoro tra i thread

della GPU potrebbe essere svolta su qualsiasi delle due dimensioni del problema. Tuttavia, la natura stessa del metodo di Chang-Cooper mostra che mentre le computazioni sulle singole particelle possono essere svolte in modo indipendente, lo stesso non si può dire delle computazioni sui diversi timestep. Infatti, l'algoritmo che implementa tale metodo presenta delle loop-carried dependencies, per le quali il calcolo dello spettro ad uno specifico timestep dipende dallo spettro calcolato per il timestep precedente. Di conseguenza, se per ogni particella si suddividessero i diversi timestep tra le unità di esecuzione disponibili, il risultato finale presenterebbe delle incongruenze dovute alla violazione delle dipendenze descritte: senza la garanzia che le iterazioni sui timestep vengano eseguite in ordine, si avrebbero delle situazioni in cui un thread cercherebbe di leggere i dati dello spettro al timestep precedente prima che questi siano stati scritti dal thread corrispondente. Inoltre, in generale si ha che N_t è molto minore di N, e quindi la suddivisione del carico di lavoro basata sui timestep non consentirebbe di sfruttare appieno le potenzialità del parallelismo GPU.

Per questo motivo, si è scelto di parallelizzare il codice di riferimento secondo un approccio simile a quello utilizzato nel parallelismo CPU già implementato: le particelle da elaborare vengono suddivise tra i vari thread a disposizione, in modo che ogni thread si occupi di svolgere le computazioni su tutti i timestep di su una singola particella. In questo modo, è possibile sfruttare in modo efficace l'elevata granularità del parallelismo offerto dalla GPU.

Una possibile limitazione di questo approccio è il fatto che l'elevata complessità delle computazioni effettuate da ciascun thread sulla propria particella può comportare problemi di prestazioni. Infatti, l'hardware della GPU è progettato per eseguire in parallelo una quantità elevata di istruzioni semplici; di conseguenza, se le operazioni eseguite dai thread sono troppo complesse, l'efficienza del parallelismo GPU potrebbe risentirne.

3.4 Implementazione

L'implementazione della versione GPU è stata svolta interamente nell'ambiente di Leonardo, il quale dispone già dei moduli necessari per lo sviluppo di codice GPU con OpenMP. Nello specifico, durante la fase di sviluppo è stato utilizzato il modulo `nvhpc`, che fornisce vari strumenti utili per la compilazione e il debugging di programmi destinati a GPU Nvidia. Tra gli strumenti inclusi in tale modulo vi è il compilatore NVC, che consente di compilare codice che fa uso di librerie directive-based per la programmazione GPU, in particolare OpenACC e OpenMP. A causa dell'elevata complessità del codice, ci si è focalizzati su una versione ridotta del programma, nella quale sono state selezionate solo alcune delle funzioni necessarie per il calcolo dello spettro. Ciò ha consentito di facilitare lo sviluppo e il debugging di una prima versione di base, che può poi rappresentare un punto di partenza per una versione completa del codice.

3.4.1 Caratteristiche della versione ridotta

La versione ridotta è stata ricavata a partire dal codice originale commentando diverse funzioni di calcolo presenti nel ciclo principale, che itera su tutte le particelle della simulazione. Sono state quindi mantenute solo le funzioni indispensabili per la corretta esecuzione del codice. In particolare, le principali funzioni richiamate nella versione ridotta sono le seguenti:

- `CRe_Norm()` e `CRp_Norm()`: Implementano l'operazione di normalizzazione dello spettro iniziale per elettroni e protoni rispettivamente.

- `CC1D_Coef()`: Imposta i coefficienti necessari per lo svolgimento del metodo di Chang-Cooper.

- `CC_1D()`: Applica il metodo di Chang-Cooper in base ai coefficienti precedentemente calcolati, risolvendo il relativo sistema tridiagonale secondo il metodo di eliminazione Gaussiana.

La funzione `CC1D_Coef()` è stata inoltre modificata in modo che utilizzi dei parametri fittizi, così da semplificare la verifica dei risultati. Tali parametri fanno sì che nel risultato della versione ridotta, lo spettro a qualsiasi timestep sia uguale allo spettro iniziale della particella. Le operazioni di calcolo vengono comunque svolte, ma risolvono ad ogni iterazione un'equazione che costituisce un'identità. Oltre a quelle riportate, che implementano le operazioni principali, si hanno anche una serie di funzioni utilizzate per il calcolo dei parametri fisici del problema:

- `c_sound()`
- `v_Alfven()`
- `B_dynamo()`
- `dv_limit()`

Dalla versione ridotta sono state quindi escluse, tra le altre cose, tutte le funzioni per il calcolo dello spettro delle radiazioni di sincrotrone.

Nella versione originale del codice è inoltre presente un'invocazione alla funzione `rand()` avente lo scopo di inizializzare con un valore casuale la variabile `on_start_index`, utilizzata per determinare quante iterazioni devono essere effettuate all'interno dei seguenti cicli, eseguiti nel corso di un'iterazione su un singolo timestep:

```
// off //
for (nsblp = 0; nsblp < on_start_index; nsblp++){
    // proton //
    CC_1D(1, np, dt_sb, CCp_off, Qpi, CRp);
    // electron //
    CC_1D(-1, npe, dt_sb, CCe_off, Inje, CRe);
}
// on //
for (nsblp = on_start_index; nsblp < on_end_index; nsblp++){
    // proton //
```

```
    CC_1D(1, np, dt_sb, CCp, Qpi, CRp);
    // electron //
    CC_1D(-1, npe, dt_sb, CCe, Inje, CRe);
}
// off //
for (nsblp = on_end_index; nsblp < N_subloop; nsblp++){
    // proton //
    CC_1D(1, np, dt_sb, CCp_off, Qpi, CRp);
    // electron //
    CC_1D(-1, npe, dt_sb, CCe_off, Inje, CRe);
}
```

In questi cicli viene invocata la funzione `CC_1D()` per il calcolo dello spettro di protoni ed elettroni. Anche il valore della variabile `on_end_index` è determinato in modo casuale, in quanto corrisponde a: `on_start_index + N_on`, dove `N_on` ha un valore intero e non nullo. La funzione `rand()`, tuttavia, non è thread-safe, e ciò vuol dire che il suo utilizzo all'interno di una regione parallela non è sicuro. Per questo motivo, tale funzione è stata momentaneamente esclusa dal codice, in modo che la variabile `on_start_index` venga inizializzata sempre al valore 0. La conseguenza di questa modifica è il fatto che il numero di iterazioni dei cicli che invocano `CC_1D()` diventa deterministico, ma ciò non causa problemi dal punto di vista dei risultati ottenuti.

3.4.2 Inserimento delle direttive

Lo sviluppo della versione GPU ha richiesto l'inserimento nel codice delle opportune direttive OpenMP per il calcolo GPU. Nel codice originale erano già incluse delle direttive OpenMP, necessarie per implementare il parallelismo CPU a memoria condivisa. In questa fase, quindi, le direttive già presenti nel codice sono state sostituite con quelle necessarie per la programmazione GPU.

Suddivisione del carico di lavoro

Per far sì che le computazioni sulle particelle vengano svolte sulla GPU, il ciclo principale è stato racchiuso all'interno della direttiva `#pragma omp target {...}`, che consente di trasferire l'esecuzione sulla GPU. La strategia di parallelismo scelta prevede che le iterazioni del ciclo principale vengano distribuite tra i vari thread della GPU, in modo che ogni thread svolga le computazioni di una singola iterazione. A questo scopo, in corrispondenza del costrutto `for` del ciclo principale, è stata inserita la direttiva:

```
#pragma omp teams distribute parallel for private(i, j)
```

Tale direttiva si compone di diversi costrutti utili a specificare il tipo di parallelismo che si vuole ottenere. In particolare:

- `teams` mette in esecuzione più team indipendenti, e all'interno di ogni team alloca un certo numero di thread.
- `distribute` suddivide le iterazioni del ciclo tra i vari team.
- `parallel for` specifica che i team allocati devono eseguire le varie iterazioni in parallelo e in qualsiasi ordine.
- `private(i, j)` specifica che le variabili i e j dichiarate fuori dalla regione parallela devono essere private, e quindi ciascun thread deve averne una copia locale.

Queste direttive fanno sì che OpenMP generi un'apposita funzione, detta funzione kernel, che implementa sulla GPU le operazioni effettuate nella regione parallela, secondo il parallelismo specificato. Il solo inserimento di tali direttive, tuttavia, non è sufficiente per il corretto funzionamento del codice.

Mapping dei dati

Oltre a comunicare ad OpenMP quali porzioni di codice devono essere parallelizzate, è necessario specificare come dev'essere svolto il mapping di

tutte le strutture dati utilizzate in fase di calcolo. Le strutture dati che devono essere mappate includono:

- Gli array di input.
- Gli array di output.
- Gli array e le strutture dati di supporto.

Per gestire il mapping, in corrispondenza del costrutto `target` sono state inserite delle clausole `map`, che descrivono le politiche di mapping da utilizzare per ciascuna variabile richiamata nella regione parallela. L'inserimento delle clausole `map` proprio in questo punto del codice fa sì che il mapping rimanga valido durante tutta l'esecuzione della regione parallela, limitando il numero di trasferimenti da host a device e viceversa.

Come già spiegato, la politica di mapping di default è `tofrom`. Per evitarlo, insieme alle clausole `map` è stata aggiunta la clausola `defaultmap` nel modo seguente:

```
defaultmap(none:aggregate) defaultmap(none:pointer)
```

Questa clausola consente di sovrascrivere la politica di default utilizzata per uno specifico tipo di dato. In questo caso, ai tipi aggregati e ai puntatori è stato assegnato il valore `none`, che forza il programmatore a specificare esplicitamente la politica di mapping per ogni variabile di un dato tipo, generando un errore in fase di compilazione se una variabile non è stata mappata.

Agli array di input è stato applicato un mapping di tipo `to`, così che i dati al loro interno vengano trasferiti solo all'inizio della regione parallela. Gli array di output, invece, vengono inizializzati all'esterno della regione parallela, e quindi fanno uso della politica `tofrom`, in modo che sulla GPU vengano copiati i dati inizializzati. Spostando l'inizializzazione all'interno della regione parallela, tuttavia, si potrebbe utilizzare la politica `from`, migliorando l'efficienza del mapping. Per gestire correttamente il mapping degli array dinamici è stata utilizzata la sintassi:

```
map(to: temp_node[0:INPUT_SIZE])
```

che consente di specificare esplicitamente la dimensione dell'area di memoria allocata. Nell'esempio mostrato, `temp_node` è il puntatore all'array di input contenente la temperatura delle particelle, e `INPUT_SIZE` è il numero di elementi da mappare. Il primo valore presente tra parentesi quadre (in questo caso 0) specifica l'offset della porzione di array da mappare rispetto al primo elemento dell'array. Un offset di 0 specifica che l'area di memoria mappata deve partire esattamente dall'elemento con indice 0.

Per quanto riguarda i dati di appoggio, essi sono gestiti attraverso `struct` di diversi tipi:

- `CRspectrum`: Contiene due scalari `double` e 6 array statici di `double` di lunghezza pari a M, e quindi 128.
- `ChangCooper`: Contiene tre variabili di tipo puntatore a `double`, utilizzate nel codice per l'indirizzamento di altrettanti array di 128 elementi.
- `FPloss`: Contiene due scalari `double` e tre puntatori a `double`, anch'essi utilizzati per gestire degli array di 128 elementi.

Oltre a questi `struct`, il codice si avvale dei seguenti array di `double`, aventi una lunghezza di 128:

- `Qpi`
- `Qepri`
- `Qe_buff`
- `Inje`
- `epsSyn`
- `epsgamma`

Questi sono solo gli array di appoggio utilizzati nella versione ridotta: il codice completo utilizza altri 9 array con le stesse caratteristiche, insieme a 4 array bidimensionali gestiti mediante delle variabili di tipo `double**`, cioè doppi puntatori a `double`, e un array tridimensionale gestito con un triplo puntatore a `double`. Nel codice originale, le strutture dati elencate vengono dichiarate ed inizializzate subito prima del ciclo principale, e sono incluse all'interno della regione parallela CPU. Ciò vuol dire che, nel codice di riferimento, ogni thread OpenMP possiede la propria copia di tutte le strutture dati ausiliarie. Essendo che tali strutture dati vengono riscritte ad ogni iterazione del ciclo principale, questa gestione locale è necessaria per evitare race condition. Per questo motivo, nella versione GPU, le dichiarazioni delle strutture ausiliarie sono state spostate nel corpo del ciclo principale, in modo che ogni thread inizializzi le proprie strutture dati direttamente sulla GPU. Così facendo, non si ha la necessità di mappare i dati ausiliari sul device all'inizio della regione parallela. Nel corso di questa modifica, tutti gli array dinamici utilizzati sono stati convertiti in array statici, in quanto, sulla GPU, l'allocazione di memoria tramite `calloc()` non è ammessa, mentre l'utilizzo di `malloc()` può essere problematico in termini di prestazioni. Le uniche strutture ausiliarie la cui dichiarazione non è stata spostata sulla GPU sono `CRproton` e `CRelectron`, di tipo `CRspectrum`. Queste strutture, infatti, vengono inizializzate tramite l'invocazione di tre funzioni apposite, presenti prima del ciclo principale. Per evitare di richiamare tali funzioni all'interno del kernel GPU, si è scelto di lasciarle all'esterno della regione parallela e mappare in modo esplicito le due strutture dati, aggiungendole alla clausola `map(to:)`. Inoltre, vista la necessità di mantenere private tali strutture, nella direttiva con `teams` è stata aggiunta la clausola `firstprivate(CRproton, CRelectron)`. Tale clausola fa sì che ogni thread erediti una copia locale e già inizializzata delle due strutture.

Funzioni invocate sulla GPU

Il calcolo dello spettro all'interno del ciclo principale fa uso di diverse funzioni definite negli altri moduli dell'applicazione, ed elencate in Sottosezione 3.4.1. Per far sì che tali funzioni possano essere invocate all'interno del codice GPU, è necessario comunicare ad OpenMP la necessità di generare anche per esse una versione kernel. Ciò viene fatto inserendo delle specifiche direttive in corrispondenza delle dichiarazioni delle funzioni interessate. In particolare, le dichiarazioni devono essere racchiuse tra la coppia di costrutti `declare target` e `end declare target`. Nel codice di riferimento, tali direttive sono state inserite all'interno dei file di intestazione dei moduli utilizzati.

3.4.3 Refactoring del codice

L'inserimento delle direttive OpenMP descritte non è sufficiente per il corretto funzionamento della versione GPU, in quanto il codice originale presenta alcune caratteristiche problematiche per l'esecuzione sul device. Oltre all'aggiunta delle direttive, quindi, lo sviluppo della versione GPU ha richiesto una fase di refactoring del codice di riferimento. Di seguito, sono illustrate tutte le modifiche apportate in questa fase.

Non contiguità degli array

Nel codice originale, gli array utilizzati per input e output sono array dinamici bidimensionali, il cui indirizzamento in memoria è basato su variabili di tipo `double**`. L'allocazione di questi array con `malloc()` e `calloc()` viene quindi effettuata in due passaggi:

1. Allocazione di un array di puntatori a `double` contenente un numero di elementi pari al numero di righe dell'array bidimensionale.

2. Per ogni puntatore nel primo array allocato, allocazione di un array di `double` avente lunghezza pari al numero di colonne nell'array bidimensionale.

Sebbene questo metodo di allocazione risulti comodo dal punto di vista dell'utilizzo degli array, esso risulta problematico se tali array devono essere elaborati da una GPU, in quanto non assicura che lo spazio di memoria allocato sia contiguo. Infatti, la memoria allocata con una singola chiamata a `malloc()` o `calloc()` è contigua, ma non si può dire lo stesso per la memoria allocata tramite più chiamate successive. Quando si opera su una GPU, lavorare su aree di memoria contigue è fondamentale sia per rendere più efficienti gli accessi in memoria, sia per effettuare correttamente il mapping dei dati con OpenMP. Durante il mapping di un array dinamico con la clausola `map`, infatti, OpenMP assume che la memoria da mappare sia contigua. In caso contrario, il mapping viene effettuato su aree di memoria non inizializzate, generando problemi di accesso durante l'esecuzione. Inoltre, l'architettura della GPU gestisce gli accessi in memoria leggendo blocchi di dati contigui, quindi il fatto che la memoria sia frammentata peggiora notevolmente l'efficienza del codice.

Per questo motivo, il codice è stato modificato in modo che ogni array venga gestito mediante un solo puntatore a `double`, e la memoria necessaria venga allocata in modo contiguo con una sola chiamata a `malloc()`. Questo approccio comporta anche una diversa modalità di accesso agli array: se si vuole accedere all'elemento a riga `i` e colonna `j` di un array contiguo A, essendo A monodimensionale la sintassi da utilizzare non è `A[i][j]`, bensì `A[i * NUM_COLS + j]`, dove `NUM_COLS` è il numero di colonne dell'array. Tutti gli accessi effettuati ad array di input o output nel codice sono stati quindi modificati per rispettare tale sintassi.

Utilizzo di VLA

Un altro aspetto problematico del codice di riferimento è l'utilizzo, all'interno di alcune funzioni di calcolo, di array di appoggio aventi lunghezza

variabile, cioè la cui dimensione è determinata a runtime in base al valore di una variabile di input. Questi array prendono il nome di Variable Length Array (VLA), e sono ammessi in alcuni standard del linguaggio C, tra cui il C17 con estensioni GNU, utilizzato per la compilazione del codice di riferimento. Quando si scrive codice per la GPU, tuttavia, il loro utilizzo non è consentito, in quanto la generazione delle funzioni kernel richiede che la dimensione degli array locali a ciascun thread sia specificata a tempo di compilazione. Questo vincolo consente al compilatore di conoscere a priori la quantità di memoria che dovrà essere riservata a ciascun thread. Nel codice di riferimento, i VLA vengono utilizzati all'interno delle funzioni `Chang_Cooper()` e `Coef_CC()`, invocate rispettivamente all'interno di `CC_1D()` e `CC1D_Coef()`. Entrambe le funzioni determinano la dimensione dei VLA utilizzati in base al valore di un parametro di input che specifica il numero di bin su cui svolgere le computazioni. Nella versione GPU, questi VLA sono stati sostituiti con array aventi una dimensione fissa. Tale conversione non altera il risultato del codice, in quanto, nel `main`, il numero di bin che si utilizza è determinato con la costante `np` definita tramite macro, il cui valore è noto già in fase di preprocessing. Per eliminare i VLA è stato, quindi, sufficiente modificare la loro dichiarazione, sostituendo il parametro di input con il valore definito nella macro. Questa modifica è necessaria per il funzionamento del codice sulla GPU, ma introduce una limitazione legata al fatto che il numero di bin su cui lavorano `Chang_Cooper()` e `Coef_CC()` non può più essere determinato a tempo di esecuzione, ma solo a tempo di compilazione.

Utilizzo di variabili e costanti esterne

Come già descritto in Sottosezione 2.1.1, il codice legge i parametri fisici del problema da un file di configurazione passato in input. Durante la fase di lettura, tali parametri vengono memorizzati in variabili definite in un modulo apposito, le quali vengono poi utilizzate dagli altri moduli sotto forma di variabili di tipo `extern`. In modo analogo vengono gestite anche le costanti fisiche utilizzate. Questo approccio è problematico dal punto

di vista dell'implementazione GPU, in quanto non consente ad OpenMP di capire quali tra quelle variabili vengono utilizzate all'interno della regione parallela. Come risultato, il compilatore non riconosce tali variabili nelle sezioni di codice GPU in cui sono richiamate, e quindi genera una serie di errori. Per sistemare questo problema, le funzioni invocate sulla GPU sono state modificate in modo che tutte le variabili o costanti esterne in esse utilizzate vengano specificate nella loro signature. In questo modo, ogni invocazione di una di queste funzioni specifica in modo esplicito tutte le variabili o le costanti che verranno utilizzate al suo interno, consentendo ad OpenMP di mapparle correttamente sulla GPU.

3.4.4 Compilazione

Nel corso dello sviluppo, la compilazione del codice GPU è stata svolta con due compilatori diversi: NVC e GCC. Entrambi i compilatori, infatti, supportano le direttive OpenMP per la programmazione sulla GPU, ma hanno caratteristiche diverse. GCC gode di una maggiore portabilità, in quanto consente di generare codice sia per GPU AMD che per GPU Nvidia. NVC è invece limitato alle schede Nvidia, ma ha un output più informativo rispetto a GCC, e per questo risulta più utile in fase di sviluppo. Se si lavora in ambiente Nvidia, entrambi i compilatori convertono le direttive OpenMP in codice CUDA. Le modalità di compilazione sono leggermente diverse a seconda del compilatore che si utilizza.

Compilazione con GCC

La compilazione con GCC della versione GPU avviene in modo simile alla versione CPU. Innanzitutto, è necessario caricare i moduli necessari nell'ambiente di Leonardo, in particolare:

- `openmpi`: contiene le librerie di MPI e il compilatore MPICC, un wrapper di GCC che supporta la compilazione per MPI.

- `hdf5`: contiene i moduli necessari per il funzionamento della libreria HDF5.
- `gsl`: contiene GSL, una libreria per il calcolo scientifico utilizzata nel codice di riferimento.

Caricati i moduli, la compilazione avviene attraverso il comando `mpicc`. Per far sì che il codice compili correttamente sulla GPU, è necessario aggiungere l'opzione `-foffload=-lm`, che seleziona la versione kernel delle funzioni matematiche contenute nella libreria cmath.

Compilazione con NVC

Come già descritto, NVC fa parte del modulo `nvhpc` di Nvidia. Per questo motivo, il suo utilizzo nell'ambiente di Leonardo richiede di caricare le versioni compatibili con tale pacchetto dei moduli elencati in precedenza. Per compilare con NVC è necessario utilizzare il relativo comando `nvc`. La riga di comando di NVC non differisce molto rispetto a GCC, ma necessita di alcune opzioni aggiuntive che specificano il tipo di target su cui si vuole compilare:

```
-mp=gpu -target=gpu -Minfo=mp
```

In particolare, `-mp=gpu` e `-target=gpu` comunicano al compilatore la presenza nel codice di direttive OpenMP per la GPU, e `-Minfo=mp` fa sì che nell'output di compilazione vengano mostrate le informazioni sulla parallelizzazione effettuata tramite OpenMP.

3.5 Esecuzione del codice e debugging

Per effettuare i test di esecuzione, il codice è stato compilato con entrambi i compilatori precedentemente descritti. Tuttavia, a causa di un problema con la gestione del modulo HDF5 da parte di NVC, i test sono stati effettuati esclusivamente sulla versione compilata con GCC. Nel corso dei test, è emerso

un problema durante l'esecuzione per il quale il codice GPU terminava con i seguenti messaggi di errore:

```
libgomp: cuCtxSynchronize error: an illegal memory access
was encountered

libgomp: cuMemFree_v2 error: an illegal memory access
was encountered
```

Per investigare la causa di questo errore è stato utilizzato il comando `compute-sanitizer`, incluso nel pacchetto `nvhpc`, che, se usato per lanciare un eseguibile, consente di visualizzare in modo dettagliato tutti gli errori generati durante l'esecuzione del codice CUDA presente al suo interno. Eseguendo `compute-sanitizer` sulla versione GPU, l'output mostrava un errore di stack overflow per ciascun thread allocato da OpenMP. Di conseguenza, sono state effettuate diverse ipotesi sulla possibile causa di tale errore:

- Presenza di un errore di accesso in memoria in uno degli array utilizzati nella regione parallela.
- Presenza di un errore nel mapping dei dati sulla GPU.
- Eccessiva complessità del codice, per cui la quantità elevata di variabili locali e invocazioni di funzione comporterebbe un utilizzo eccessivo dello stack di ciascun thread.

La prima ipotesi è stata esclusa attraverso un test che ha previsto la rimozione dal codice di tutte le direttive di programmazione GPU, in modo da verificare se le modifiche apportate in fase di refactoring avessero introdotto un errore di gestione della memoria. Questo test non ha evidenziato errori, in quanto i risultati ottenuti sono coerenti con quelli del codice di riferimento. Anche eseguendo il codice con `valgrind`, un comando per il monitoraggio dell'utilizzo della memoria da parte di un eseguibile, non sono stati trovati memory leak o problemi di accesso in memoria.

Esclusa l'ipotesi di un errore nel refactoring, si è cercato di capire se il problema fosse dovuto all'eccessiva complessità del codice GPU. A questo scopo, il programma è stato testato commentando parti di codice della regione parallela, in modo da escluderle dalla computazione. Da questi test, è emerso che se si commenta una parte consistente del ciclo principale, l'esecuzione riesce a terminare correttamente. Questo risultato sembra quindi avvalorare l'ipotesi di un codice troppo complesso. Si è deciso quindi di controllare l'effettivo utilizzo dello stack mediante una specifica opzione di compilazione disponibile in NVC: `-gpu=ptxinfo`, che consente di visualizzare la quantità di memoria occupata sullo stack da ciascuna funzione kernel generata. Analizzando l'output di compilazione, si è visto che l'utilizzo dello stack da parte di diverse funzioni, incluso il kernel del main, risultava piuttosto elevato. Uno dei fattori che contribuiva ad aumentare l'occupazione dello stack è la grande quantità di array statici utilizzati come array di supporto all'interno della regione parallela. La memoria occupata da tali array, infatti, viene allocata interamente sullo stack locale di ciascun thread, e non viene liberata fino alla fine della regione parallela. Per cercare di ridurre l'utilizzo dello stack sono state quindi apportate delle modifiche al codice, in modo da sostituire gli array statici locali con array globali dichiarati sulla CPU e poi mappati sulla GPU. Per ognuno degli array statici di appoggio (aventi una lunghezza di 128) è stato quindi definito un array globale dinamico con lunghezza pari a $128 \times N$. In questo modo, ogni thread ha a disposizione dei sottoarray di 128 elementi che può utilizzare liberamente per svolgere le computazioni sulla propria particella. Per quanto riguarda il mapping, gli array globali così definiti sono stati mappati secondo la politica `alloc`, in modo da allocare la memoria necessaria sulla GPU senza effettuare trasferimenti superflui di dati. Verificando l'output di NVC, si è notato che queste modifiche hanno contribuito a ridurre notevolmente l'utilizzo dello stack da parte del kernel del main. Infatti, la quantità di stack allocata per il main su ogni thread è passata da più di 35.000 byte a poco meno di 400. Nonostante ciò, tale soluzione non è stata sufficiente per eliminare il problema di stack

overflow.

Un ulteriore tentativo effettuato in questa fase ha previsto la rimozione dalla clausola `map` delle strutture dati `CRproton` e `CRelectron`, allo scopo di capire se il problema sia dovuto ad un mapping errato di questi `struct` da parte di OpenMP. In questo test, le due strutture dati sono state decomposte negli array in esse contenuti, che sono stati mappati singolarmente. All'interno della regione parallela sono poi state inserite delle istruzioni per ricostruire le due strutture dati, a partire dai valori degli array mappati. Anche in questo caso, le modifiche effettuate non hanno consentito di risalire alla causa principale del problema. Tuttavia, questo risultato non consente di escludere del tutto l'assenza di errori nel mapping degli `struct` da parte di OpenMP. Per approfondire questo aspetto, sarebbero quindi necessari degli ulteriori test.

Nel corso dei test effettuati non è stato possibile escludere l'ipotesi secondo cui l'errore sarebbe causato da un problema nel mapping dei dati. Infatti, la verifica di tale ipotesi richiederebbe un'analisi più approfondita e di basso livello della gestione del mapping da parte di OpenMP. Tutto ciò, tuttavia, va oltre gli obiettivi di questa tesi.

Capitolo 4

Conclusioni

Il lavoro presentato in questa tesi ha consentito di raggiungere con successo gli obiettivi prefissati. È stato, infatti, possibile analizzare su diversi casi di test le prestazioni di entrambi i tipi di parallelismo CPU implementati nel codice considerato. È stata poi realizzata una versione GPU secondo l'approccio a direttive di OpenMP, consentendo di valutare pregi e difetti dell'utilizzo di questa libreria all'interno di un codice di produzione complesso e non facilmente ottimizzabile con metodi convenzionali.

4.1 Considerazioni sui risultati ottenuti

La valutazione delle prestazioni effettuata sul codice di riferimento ha mostrato che l'utilizzo di OpenMP ed MPI per l'ottimizzazione sulla CPU offre già un buon grado di parallelismo. I test hanno inoltre mostrato che i due approcci risultano molto simili tra loro in termini di prestazioni, e le principali differenze sono causate dalle caratteristiche intrinseche dei due diversi paradigmi di parallelizzazione adottati. Nonostante questa somiglianza nell'efficienza dei due approcci, è stato comunque possibile determinare quale risulta più vantaggioso in specifici casi. In particolare, si è osservato che l'approccio di OpenMP ha una maggiore efficacia in termini di strong scaling, mentre l'approccio MPI, con un numero di nodi elevato, offre prestazioni

migliori nei casi in cui il numero di unità di esecuzione viene incrementato proporzionalmente alla dimensione del problema.

Per quanto riguarda la versione GPU, si è osservato che, nonostante le premesse, l'approccio a direttive offerto da OpenMP non è immediato, e richiede potenzialmente molto lavoro per adattare il codice esistente all'esecuzione sul device. Infatti, il solo inserimento delle direttive fornite dalla libreria non è sufficiente ad ottenere un codice in grado di eseguire o persino di compilare correttamente, e dev'essere quindi accompagnato da una serie di modifiche al codice di partenza. Tutto il lavoro svolto nella fase di refactoring illustrata in precedenza ha quindi l'obiettivo di supportare l'approccio a direttive nel modo migliore possibile, ma non è stato sufficiente a consentire il corretto funzionamento della versione GPU. Uno degli aspetti più problematici dell'utilizzo di OpenMP è il fatto che l'elevato livello di astrazione su cui si basa rende difficile determinare l'effettiva modalità di ottimizzazione utilizzata. Infatti, se confrontata con librerie più di basso livello, come CUDA, ROCm e OpenCL, OpenMP offre al programmatore un minor controllo sulle operazioni che vengono effettuate sulla GPU, complicando notevolmente la diagnosi e la risoluzione di eventuali problemi riscontrati. D'altra parte, in molti casi OpenMP consente di ottenere un parallelismo GPU attraverso l'aggiunta di molte meno righe di codice rispetto alle librerie sopra citate. Inoltre, a differenza di queste librerie, più legate all'hardware utilizzato, OpenMP vanta un'elevata portabilità, grazie all'approccio più ad alto livello. OpenACC presenta caratteristiche simili ad OpenMP, ma ha il vantaggio di essere una tecnologia più consolidata, e di conseguenza più ottimizzata in determinati contesti. Tuttavia, attualmente OpenMP sta subendo un forte sviluppo, e in futuro potrebbe riuscire a raggiungere o anche superare OpenACC in termini di efficienza. Possiamo quindi concludere che OpenMP è un paradigma di programmazione in grado di facilitare notevolmente la realizzazione di codice GPU portabile, ma il suo utilizzo può essere problematico, e per nulla semplice, se il programma da ottimizzare ha un elevato grado di complessità.

4.2 Sviluppi futuri

4.2.1 Valutazione delle prestazioni del codice CPU

Sebbene la valutazione delle prestazioni effettuata in questa tesi abbia già consentito di ricavare molte informazioni sullo stato del parallelismo CPU, esistono ulteriori test che si possono svolgere per migliorare l'accuratezza di tali informazioni. Uno dei test che si possono effettuare è la misurazione delle prestazioni che si ottengono combinando i due tipi di parallelismo implementati, quindi allocando diversi nodi MPI, ognuno su un nodo fisico del cluster, con più core OpenMP su ogni nodo. In questo modo, sarebbe possibile valutare l'efficienza del programma quando il parallelismo al suo interno viene sfruttato al massimo delle potenzialità. Per migliorare la comprensione della weak scaling efficiency del codice, si potrebbe inoltre effettuare un ulteriore test di scalabilità in cui il carico di lavoro viene modificato incrementando progressivamente il numero di particelle, invece che il numero di timestep.

4.2.2 Versione GPU

Un primo sviluppo possibile della versione GPU realizzata è rappresentato dalla risoluzione del problema di stack overflow riscontrato. Ciò richiederebbe un'analisi più approfondita delle funzionalità di OpenMP, soprattutto per quanto riguarda l'allocazione di memoria su ciascun thread. A tal scopo, ci si potrebbe servire di strumenti di debugging più avanzati rispetto a quelli utilizzati nel corso del lavoro presentato, in modo da monitorare con maggior precisione le modalità di gestione della memoria utilizzate da OpenMP. Uno di questi è CUDA-GDB, un'applicazione offerta da Nvidia che consente di effettuare il debug di codice CUDA, compreso quello generato da OpenMP. Tale strumento è già presente nell'ambiente di Leonardo, all'interno del modulo `nvhpc`. Oltre ad utilizzare debugger come questo, si potrebbe tentare di eseguire il codice nella sua versione compilata con NVC. Quest'ultimo compilatore, infatti, è più ottimizzato rispetto a GCC per quanto riguarda la

compilazione su GPU Nvidia. L'utilizzo di NVC potrebbe quindi agevolare le operazioni di debug o anche rimuovere del tutto il problema, nel caso in cui questo sia dovuto ad un bug di GCC. Inoltre, tra i test che si possono effettuare per identificare la causa del problema, si hanno:

- L'esecuzione del codice su dati fittizi di piccole dimensioni, in modo da ridurre al minimo la quantità di memoria richiesta.
- La realizzazione di una versione minimale del codice GPU, nella quale si limita il più possibile la quantità di dati mappati sul device. In tale versione, si potrebbe ad esempio utilizzare un solo array di input e uno di output, in modo da ridurre il numero di trasferimenti di dati. Naturalmente, il risultato di tale versione non sarebbe corretto, ma la sua esecuzione consentirebbe di capire se il problema sia legato al mapping dei dati.
- La realizzazione di un codice di prova con lo scopo di verificare che OpenMP effettui in modo corretto il mapping delle strutture dati specificate nella clausola `map`. In tale codice, si potrebbero creare delle `struct` contenenti degli array statici, inizializzati con un certo valore. Tali `struct` possono essere poi mappate sul device, in modo da verificare se i dati immagazzinati nei loro array vengono correttamente copiati sulla GPU in fase di mapping.

È, infine, necessario tenere presente che il codice del ciclo parallelizzato è particolarmente complesso, e quindi il problema potrebbe essere legato proprio all'eccessiva complessità delle operazioni al suo interno. In tal caso, la sua risoluzione richiederebbe di modificare la strategia di parallelizzazione adottata oppure la struttura stessa del codice, in modo che ogni thread possa svolgere operazioni più semplici.

Un altro possibile sviluppo è la progressiva estensione dell'implementazione GPU all'intero codice, includendo tutte le funzioni del ciclo principale che non sono state prese in considerazione nella versione ridotta. L'estensione prevede di effettuare sul resto del codice una fase di refactoring simile

a quella riportata in Sottosezione 3.4.3, per poi aggiungere le direttive aggiuntive necessarie per il corretto funzionamento del programma sulla GPU. Un dettaglio importante da considerare in questa fase è l'inclusione nel ciclo parallelizzato della funzione `rand`, richiamata in ogni thread per generare numeri casuali. Tale funzione, infatti, non è thread-safe, e per questo motivo, nell'implementazione GPU completa, va sostituita con un metodo di generazione di numeri casuali compatibile con l'esecuzione sul device. Un approccio possibile potrebbe essere l'adozione di cuRAND, una libreria di CUDA che consente di generare sequenze casuali sulla GPU. Tuttavia, essendo cuRAND legata ad Nvidia, il suo utilizzo limiterebbe la portabilità del codice realizzato, impedendo di sfruttare alcuni dei vantaggi offerti dal paradigma di OpenMP.

Realizzata la versione completa dell'implementazione GPU, sarà poi possibile svolgere la valutazione delle sue prestazioni ed effettuare un confronto con il precedente parallelismo CPU, in modo da determinare l'efficienza dell'approccio offerto da OpenMP. Nel caso in cui la strategia di parallelizzazione illustrata in Sezione 3.3 risulti poco efficiente, a causa dell'elevata complessità del ciclo principale, in futuro si potrà tentare di ottimizzare il codice secondo un parallelismo a granularità più fine, in cui ogni thread esegue operazioni più semplici, per meglio assecondare le caratteristiche dell'architettura hardware della GPU. A tal proposito, può essere utile una fase preliminare di misurazione del tempo di esecuzione di ognuna delle funzioni di calcolo richiamate nel ciclo principale, con lo scopo di identificare quali sezioni del codice devono essere ottimizzate con maggiore priorità.

Appendice A

Lavorare nell'ambiente di Leonardo

In questa appendice sono illustrate informazioni utili per lavorare all'interno dell'ambiente di esecuzione del supercomputer Leonardo. Le informazioni riportate sono tratte dalla documentazione sull'utilizzo dei sistemi HPC messa a disposizione dal CINECA, e alcune di esse possono essere soggette a modifiche nel tempo. Si rimanda quindi a tale documentazione per indicazioni più precise riguardo gli argomenti qui trattati.

A.1 Cenni sull'architettura di Leonardo

Leonardo è un Cluster di Processori, e in quanto tale è composto da un numero elevato di nodi indipendenti interconnessi tra loro mediante una rete ad alte prestazioni. I nodi di Leonardo sono divisi in due partizioni aventi caratteristiche diverse:

Partizione Booster: Partizione dotata di GPU basate sull'architettura Ampere A100 di Nvidia. Le specifiche di un nodo in questa partizione sono indicate in Tabella 2.1.

Partizione DCGP (Data Centric General Purpose): Partizione priva di GPU, ma in cui ogni nodo è dotato di due processori Intel Sapphire Rapids che offrono complessivamente 112 core per nodo.

La partizione utilizzata nel corso del lavoro svolto in questa tesi è la Booster, e per questo motivo le sue specifiche sono riportate in modo più dettagliato.

L'accesso ai nodi di calcolo delle due partizioni è gestito da degli appositi nodi di login, ai quali ci si connette da terminale mediante il protocollo SSH. Attraverso questi nodi, è possibile accedere al file system del cluster ed inviare comandi ai nodi di calcolo. Un altro tipo di nodi è costituito dai Datamovers, che hanno il compito di gestire il trasferimento di grandi quantità di dati tra il cluster e l'esterno.

A.2 Utilizzo del file system

La gestione dello spazio di archiviazione di Leonardo si basa su un file system distribuito che permette ai diversi nodi del cluster di condividere gli stessi dati. Ciò consente di mantenere la struttura logica del file system consistente su tutti i nodi. Lo spazio di archiviazione è diviso in diverse aree, che possono essere classificate in vari modi. In particolare, le aree possono essere:

temporanee: I dati al loro interno sono accessibili solo per un certo periodo di tempo, dopo il quale vengono cancellati.

permanenti: I dati immagazzinati sono accessibili per tutta la durata del progetto, e fino a sei mesi dopo la sua fine.

Inoltre, le aree possono essere classificate in base alla visibilità dei dati al loro interno:

user specific: I dati sono accessibili solo ad un utente specifico.

shared: I dati sono condivisi tra tutti gli utenti di uno stesso progetto.

open: I dati sono accessibili a tutti gli utenti del cluster.

Tra le aree presenti nel file system di Leonardo si hanno:

$HOME La home directory dell'utente, un'area permanente e user specific di dimensioni ridotte e soggetta a backup giornalieri.

$WORK Un'area permanente e shared con 1TB di memoria a disposizione.

$FAST Come $WORK, ma utilizza dischi più veloci, che consentono migliori prestazioni nelle operazioni di input/output.

$SCRATCH Un'area temporanea e user specific, concepita per i file temporanei usati dalle applicazioni.

Ogni area può essere acceduta attraverso la corrispondente variabile di ambiente nella shell di lavoro, che contiene il suo percorso assoluto. Ciò consente di spostarsi in una specifica area passando al comando `cd` il valore della variabile corrispondente, ad esempio:

```
cd $FAST
```

La modalità di utilizzo raccomandata per queste aree prevede di salvare su $HOME i file importanti di piccole dimensioni, come script, codice o eseguibili, e mantenere su $WORK i file di grandi dimensioni, ad esempio quelli contenenti i dati da elaborare o i risultati di un'esecuzione che si intende conservare. Inoltre, $WORK è shared, e quindi può anche essere utilizzata per immagazzinare i file che si vuole condividere con gli altri collaboratori del progetto. Essendo più lenta, tuttavia, quest'area non è consigliata per l'utilizzo in lettura/scrittura da parte dei programmi in esecuzione. Per questo motivo, durante l'esecuzione si raccomanda di utilizzare $FAST per immagazzinare i dati necessari e i risultati prodotti, in modo che le applicazioni possano accedervi in modo efficiente.

A.3 Scheduler e creazione di job

L'esecuzione di codice sui nodi di Leonardo avviene secondo modalità diverse da quelle convenzionali: il cluster è utilizzato contemporaneamente da un gran numero di utenti, e per questo motivo si serve di un sistema di job scheduling per gestire l'allocazione delle risorse hardware necessarie a ciascun programma. Per eseguire un programma su Leonardo è quindi necessario definire un opportuno job, che specifica i comandi da eseguire sul cluster e le risorse necessarie per l'esecuzione. Una volta creato, il job viene inserito dallo scheduler in una coda di attesa, e successivamente messo in esecuzione non appena le risorse richieste diventano disponibili. Lo scheduler utilizzato dal cluster è Slurm (Simple Linux Utility for Resource Management), un sistema open-source con un'elevata scalabilità. L'allocazione di job tramite Slurm può avvenire secondo due modalità:

Modalità batch: Alloca un job definendo l'insieme di comandi che devono essere eseguiti e le risorse necessarie per eseguirli. Una volta avviato il job, l'insieme dei comandi specificati al suo interno non può essere modificato. Questa modalità è usata per le esecuzioni di produzione e per test che comportano tempi di esecuzione elevati.

Modalità interattiva: Alloca le risorse richieste e fornisce all'utente una console interattiva dalla quale è possibile inviare comandi al cluster. Questa modalità è solitamente utilizzata per test veloci e debugging.

Per utilizzare la modalità batch è necessario scrivere un apposito script che specifica sia i comandi da eseguire, sia i parametri di allocazione del job. Questi ultimi sono specificati mediante una serie di direttive `#SBATCH`, e comunicano allo scheduler una serie di informazioni tra cui:

- Le risorse necessarie.
- I file in cui ridirezionare standard output e standard error del programma.

- Il tempo massimo di allocazione.

Mediante le direttive si ha anche la possibilità di specificare la mail dell'utente, in modo che il sistema invii automaticamente le notifiche sullo stato del job.

La modalità interattiva prevede, invece, di richiedere l'allocazione delle risorse del cluster mediante il comando `salloc`. Questo comando ha una serie di opzioni che corrispondono alle direttive della modalità batch. In entrambe le modalità, per far sì che un comando venga effettivamente eseguito sui nodi del cluster, è necessario lanciarlo attraverso il comando `srun`, che accetta i seguenti parametri:

- Il nome del comando o dell'eseguibile da lanciare sul cluster.
- I parametri di input del comando o dell'eseguibile.

Tutti i comandi che non sono eseguiti con `srun` vengono eseguiti sul login node in cui si trova l'utente, invece che sul cluster.

Di seguito, è mostrato un esempio di script sbatch utilizzato per eseguire il codice di riferimento su 32 nodi MPI sfruttando 8 nodi della partizione Booster e 4 core fisici per nodo:

```
#!/bin/bash
#SBATCH --output=output.log
#SBATCH --error=error.log
#SBATCH --job-name=tracer_CPU_test_32_nodes     # Descriptive job name
#SBATCH --time=12:00:00     # Maximum wall time (hh:mm:ss)
#SBATCH --ntasks=32
#SBATCH --nodes=8       # Number of nodes to use
#SBATCH --ntasks-per-node=4
#SBATCH --cpus-per-task=1       # Number of CPU cores per task
#SBATCH --partition=boost_usr_prod      # GPU-enabled partition
#SBATCH --qos=normal        # Quality of Service
#SBATCH --mem=4G
```

```
#SBATCH --account=account_number      # Project account number
#SBATCH --mail-user=your@mail.com
#SBATCH --mail-type=ALL

module load hdf5
module load gsl
module load openmpi

srun ./tracer_mpi_eval.out params.txt 0
```

Nello script, `--ntasks` specifica il numero di nodi MPI che si vogliono allocare, `--nodes` indica il numero di nodi fisici del cluster e `--ntasks-per-node` specifica che su ogni nodo fisico devono essere eseguiti 4 nodi MPI, sotto forma di processi logicamente separati.

Il seguente, invece, è un esempio di comando `salloc` che consente di richiedere l'allocazione di un job interattivo su un singolo nodo, utilizzando una delle GPU al suo interno:

```
salloc -n 1 --mem-per-cpu=4G --gres=gpu:1 -A account_number \
    -p boost_usr_prod -q boost_qos_dbg -t 00:10:00
```

Questo comando imposta un tempo massimo di allocazione di 10 minuti, dopo il quale la console del job viene chiusa automaticamente. La limitata quantità di risorse e tempo di utilizzo richiesti fa sì che il job venga allocato in tempi brevi, consentendo di svolgere in modo rapido test di piccole dimensioni.

Il comando `squeue` consente di controllare in ogni momento lo stato della coda di scheduling, e l'opzione `-u` fa sì che vengano visualizzati solo i job richiesti dall'utente. I job già allocati possono inoltre essere gestiti mediante i comandi `scontrol` e `scancel`. `scontrol`, oltre a fornire informazioni più dettagliate su uno specifico job, dà anche la possibilità, mediante il sottocomando `hold`, di impedirne momentaneamente l'esecuzione. Un job in hold può poi essere rilasciato mediante il sottocomando `release`, in modo che possa essere messo in esecuzione. Il comando `scancel` consente invece di

cancellare uno o più job nella coda. Sia `scontrol` che `scancel` accettano come parametro aggiuntivo l'ID del job da gestire.

A.4 Gestione dei moduli

Nell'ambiente di Leonardo, i pacchetti software di terze parti sono installati secondo un meccanismo a moduli. Per poter utilizzare un determinato comando, eseguibile o libreria nel sistema, è quindi necessario caricare nel proprio ambiente di lavoro il modulo software corrispondente. La gestione dei moduli caricati avviene mediante il comando `module`, le cui funzioni sono accessibili mediante specifici sottocomandi. I principali sottocomandi sono:

`module avail` stampa una lista di tutti i moduli disponibili nell'ambiente di Leonardo.

`module load <nome modulo>` carica il modulo con il nome specificato.

`module list` consente di vedere tutti i moduli attualmente caricati nella sessione corrente.

`module unload <nome modulo>` rimuove il modulo specificato dalla sessione corrente.

`module purge` rimuove tutti i moduli caricati nella sessione corrente.

I moduli restano caricati nell'ambiente di lavoro dell'utente fino alla fine della sessione. Quindi, ad ogni nuovo accesso al sistema è necessario caricare nuovamente tutti i moduli necessari. Inoltre, quando si richiede l'esecuzione di un job sul cluster, è necessario caricare in modo esplicito tutti i moduli in esso utilizzati. Per la modalità batch, ciò viene fatto inserendo nello script i comandi `module load` prima dell'invocazione del comando principale, mentre nella modalità interattiva basta invocare tali comandi direttamente sul terminale. Il comando `modmap` è molto utile per cercare un determinato software all'interno dell'ambiente di Leonardo. Infatti, se invocato con l'opzione

`-m` seguita dal nome di un eseguibile, un comando o una libreria, stampa una lista di tutti i moduli in cui è installato il software specificato.

A.5 Consumo delle risorse

Le risorse che il cluster mette a disposizione degli utenti non sono illimitate: ogni progetto dispone di un budget espresso in numero massimo di ore di esecuzione consentite e condiviso tra tutti gli utenti che ne fanno parte. Ogni mese, gli utenti associati ad un progetto hanno a disposizione una quota di ore pari al budget totale, diviso la durata complessiva in mesi del progetto. Le risorse di cui può usufruire ciascun utente dipendono quindi dai progetti in cui partecipa. Le ore del budget sono espresse in ore CPU effettive, e non in tempo di orologio; di conseguenza, il consumo del budget da parte di ogni job è calcolato in base al numero di risorse in esso allocate, secondo la seguente formula:

$$B_H = T \times N \times R \times C \tag{A.1}$$

dove:

T è il tempo di esecuzione in ore

N è il numero di nodi allocati

R è un fattore che determina la frazione di risorse allocate per ogni nodo

C è il numero di core allocati su ogni nodo

Il fattore R, in particolare, è calcolato considerando il massimo tasso di utilizzo tra tutte le risorse a disposizione del nodo, tra cui numero di core, GPU, memoria. Ad esempio, se un job alloca tutti i core di un nodo, allora il tasso di utilizzo della CPU è pari ad 1, e quindi il nodo conta come completamente occupato anche se le altre risorse (GPU, memoria) non sono utilizzate del tutto.

La quantità di ore rimanenti nella quota mensile determina anche la priorità associata ai job all'interno della coda. In particolare, la priorità dei job

decresce linearmente con la percentuale di ore rimanenti, ed è massima all'inizio di ogni mese, quando si dispone dell'intera quota. Se la quota mensile è esaurita, i job creati vengono comunque presi in considerazione, ma la loro priorità è molto più bassa rispetto ai job degli utenti con quota rimanente.

Bibliografia

[1] W Baade F Zwicky. Cosmic Rays from Super-Novae. *Proc Natl Acad Sci U S A*, 20(5):259–263, 1934.

[2] S. Borgani A. Kravtsov. Cosmological Simulations of Galaxy Clusters. *Advanced Science Letters*, 4(2):204–227, February 2011.

[3] J.S. Chang G. Cooper. A practical difference scheme for fokker-planck equations. *Journal of Computational Physics*, 6(1):1–16, 1970.

[4] HESS Collaboration, A. Abramowski, et al. Acceleration of petaelectronvolt protons in the Galactic Centre. *Nature*, 531(7595):476–479, Mar 2016.

[5] Marzena Lapka. CMS Knowledge Transfer: Cosmic rays. CMS Collection., 2017.

[6] S. Navas et al. Review of particle physics. *Phys. Rev. D*, 110:535–556, Aug 2024.

[7] B. T. Park V. Petrosian. Fokker-Planck Equations of Stochastic Acceleration: A Study of Numerical Methods. *ApJS*, 103:255, March 1996.

[8] L. H. Thomas. Elliptic problems in linear difference equations over a network. *Watson Sci. Comput. Lab. Rept., Columbia University, New York*, 1:71, 1949.

[9] M. Turisini, M. Cestari, G. Amati. Leonardo: A pan-european pre-exascale supercomputer for HPC and AI applications. *JLSRF*, 9(1):4–5, 2024.

Ringraziamenti

Voglio innanzitutto ringraziare il mio relatore, Moreno Marzolla e il mio correlatore, Claudio Gheller, per avermi dato la possibilità di lavorare al confine tra due ambiti scientifici che mi appassionano: l'High Performance Computing e l'Astronomia, e per avermi dato l'occasione di utilizzare un cluster ad altissime prestazioni come Leonardo.

Ringrazio i miei genitori e i miei nonni, per avermi sempre amato e non avermi mai fatto mancare nulla. È grazie a voi se ora ho le conoscenze e gli strumenti necessari per far bene nella vita. Non vi potrò ringraziare mai abbastanza.

Grazie a Federico e Cristina, colleghi e ottimi amici, per aver reso questi anni di Università indimenticabili e preziosi. Siete riusciti a trasformare le lezioni da semplice apprendimento ad un momento di crescita, condivisione e socialità. Un grazie anche a tutti gli altri colleghi che mi hanno accompagnato in questo percorso.

Grazie a Tommaso e Nicola, grandi amici e "compagni" non solo di scuola, per tutto il tempo passato in vostra compagnia. In questi anni di Università, abbiamo condiviso viaggi, esperienze, sessioni di gaming, dimostrando che le nostre strade, seppur diverse, sono sempre legate.

Grazie a Claudia, per tutto l'amore che mi hai dato in questo fantastico anno insieme. Grazie per essermi stata sempre accanto, in tutti i bei momenti che abbiamo condiviso e anche in quelli più difficili. Sei la persona che riesce a darmi più calma e conforto, e te ne sono grato.