



UNIVERSITÀ DEGLI STUDI ROMA TRE

Dipartimento di Ingegneria
Corso di Laurea in Ingegneria Informatica

Tesi Di Laurea

Simulazione Fluidodinamica di processi di produzione additiva

Laureando

Edoardo Costantini

Matricola 500130

Relatore

Ing. Franco Milicchio

Anno Accademico 2019/2020

A chi non ha mai smesso di credere in me.

Ringraziamenti

Prima di procedere con la tesi mi sembra doveroso dedicare alcune parole a chi ha reso possibile la realizzazione di questo progetto. Un sentito grazie al Prof. Milicchio per avermi dato la possibilità di perseguire un percorso così illuminante, estremamente formativo e interessante, e per avermi supportato per tutta la durata del progetto con materiale utile e immensa esperienza nel settore. Ringrazio inoltre il Prof. Formica per aver supportato il progetto dall'inizio alla fine, costantemente. Infine per ultimo ma certamente non meno importante, Mauro con cui ho potuto condividere gioie e dolori durante l'implementazione del solutore numerico e che mi ha supportato fin dall'inizio nonostante le mie iniziali lacune di carattere fisico-matematiche.

Introduzione

Per stare al passo con la sempre più esigente richiesta dei sistemi manifatturieri moderni, vi è la necessità di introdurre nuovi processi di manifattura capaci di rispondere prontamente a tale richiesta. In questo contesto la manifattura additiva offre diversi vantaggi rispetto i più tradizionali processi. Tra questi notiamo soprattutto la capacità di realizzare complessi componenti in termini di geometrie e materiali utilizzati. Proprio questa libertà di operazione rende la manifattura additiva responsabile anche di un *time-to-market* notevolmente ridotto. La manifattura additiva rende il processo di creazione dei componenti totalmente digitale, scavalcando numerosi problemi della versione più tradizionale, come per esempio logistica e stoccaggio delle parti necessarie. Di conseguenza i costi di una produzione basata su manifattura additiva sono relativamente ridotti. La manifattura additiva introduce anche vantaggi in ambito riparazioni di precisioni, abbassando i costi operando direttamente su componenti danneggiati senza il bisogno di sostituzioni. Si può trarre beneficio di questi vantaggi sia se si opera in ambiti di piccola produzione che di prototipazione. La manifattura additiva è un ambito di ricerca in rapida evoluzione poiché diversi settori traggono benefici da queste tecniche additive, tra questi ricordiamo il settore automobilistico, aereospaziale, e biomedico.

Obiettivo

L'obiettivo di questo progetto di tesi è quello di riuscire a simulare correttamente, dal punto di vista numerico, uno scenario di manifattura additiva attraverso metodo di Laser Metal Deposition. Lo scenario prevede l'emissione da parte di un ugello di particelle metalliche schermate da un gas inerte che previene l'ossidazione su una superficie obiettivo; con particolare interesse verso l'interazione delle particelle con il flusso di gas,

in termini di temperatura e pressione, e con le pareti dell'ugello, in termini di deflezioni. Perseguendo questo obiettivo è stato sviluppato un simulatore numerico capace di sfruttare metodi Lagrangiani per il calcolo delle traiettorie delle particelle metalliche e metodi Euleriani per il calcolo delle proprietà del flusso di gas. Il simulatore sviluppato è inoltre in grado di produrre dei file interpretabili da popolari strumenti di visualizzazione come mostrato nel capitolo 6.

Capitoli

Di seguito una breve introduzione di ogni capitolo della presente tesi.

Capitolo 1. Introduzione ad alto livello dei vari processi di manifattura additiva odierni e una vista più dettagliata del processo LMD oggetto di questa tesi. Questo capitolo può considerarsi una estensione dell'introduzione.

Capitolo 2. Presentazione di alcuni concetti matematici fondamentali per la comprensione della tesi. In particolare vengono introdotti argomenti quali: equazioni differenziali alle derivate parziali, equazioni di Navier-Stokes, metodo degli elementi finiti, metodo Newton-Raphson. Il capitolo va inteso come una contestualizzazione e introduzione dei concetti, non vi sono quindi dimostrazioni.

Capitolo 3. Visione del settore della CFD e dei concetti chiave che lo caratterizzano. In questo capitolo vi è anche un dettaglio sui metodi Lagrangiani-Euleriani utilizzati per sviluppare il solutore oggetto della tesi. Infine alcuni esempi di applicazioni comuni per dove le simulazioni CFD trovano largo impiego.

Capitolo 4. Il solutore sviluppato fa uso di svariati strumenti tecnologici che hanno influenzato, sia positivamente che non, il lavoro svolto; questo capitolo descrive questi strumenti e il loro utilizzo all'interno della tesi.

Capitolo 5. Il quinto capitolo contiene una descrizione dettagliata del funzionamento e dell'architettura del solutore sviluppato, oggetto della tesi. Vengono presentati e descritti tutti i moduli che compongono il solutore, essi sono organizzati uno per sezione.

Capitolo 6. Questo capitolo espone i risultati raggiunti con il solutore implementato, fornendo diverse figure e render descrittive di quanto è stato fatto. I risultati sono raggruppati per area di interesse e infine mostrati nel loro insieme.

Capitolo 7. Il capitolo conclusivo descrive brevemente ciò che è stato raggiunto. Vengono inoltre discussi alcuni futuri sviluppi riguardo il progetto svolto.

Indice

Introduzione	iv
Obiettivo	iv
Capitoli	v
Indice	vii
Elenco delle figure	x
1 Manifattura additiva	1
1.1 Processo LMD	2
2 Modello Matematico	5
2.1 Equazioni differenziali alle derivate parziali	5
2.2 Equazioni di Navier-Stokes	6
2.2.1 Modello matematico	6
2.3 Metodo elementi finiti	8
2.3.1 Esempio FEM con un grado di libertà	9
2.4 Metodo Newton-Raphson	13
3 Computazione Fluidodinamica	14
3.1 Modello 3D	14
3.2 Discretizzazione del volume	15
3.2.1 Mesh	15
3.3 Solver	16
3.3.1 Condizioni al contorno	17

3.3.2	Interpolazione	17
3.3.3	Gradi di libertà	18
3.3.4	Metodi Lagrangiani-Euleriani	19
3.3.5	Ouput	20
3.4	Applicazioni comuni	21
4	Background Tecnologico	26
4.1	Linguaggi Utilizzati	26
4.1.1	Python versione 3.6	27
4.1.2	C++	27
4.2	Librerie e Framework	31
4.2.1	Intel TBB	31
4.2.2	MPI	34
4.2.3	OpenFoam	36
4.2.4	FEniCS Project	42
4.2.5	deal.II	43
4.2.6	Aspect	45
4.3	Strumenti di sviluppo	45
4.3.1	Gestione Librerie e pacchetti	45
4.3.2	IDE	46
4.3.3	Compilatori e altri strumenti	47
4.3.4	Docker	50
4.4	Strumenti di visualizzazione e meshing	53
4.4.1	Paraview	53
4.4.2	SALOME	55
5	Solutore Computazionale	57
5.1	Gestione Input e impostazioni	59
5.1.1	Struttura file di input	61
5.1.2	Implementazione	65
5.2	Log e statistiche	67
5.2.1	Implementazione	68

5.3	Interpretazione della Mesh	69
5.3.1	Implementazione	70
5.4	Navier-Stokes	71
5.4.1	Inizializzazione	71
5.4.2	Passo risolutivo	74
5.4.3	Sistemi non lineari	75
5.5	Temperatura e Laser	75
5.5.1	Temperatura	76
5.5.2	Laser	77
5.6	Particelle	78
5.6.1	Generazione	81
5.6.2	Avvezione	82
5.7	Parallelizzazione	84
5.8	Output	86
6	Risultati	88
6.1	Mesh	88
6.2	Fluido	91
6.3	Laser	94
6.4	Particelle	95
	Conclusioni e sviluppi futuri	101
	Ulteriori parallelizzazioni	101
	Bibliografia	103

Elenco delle figure

1.1	Vari processi di manifattura additiva.	2
1.2	Raffigurazione di un ugello LMD.	4
2.1	Soluzioni dell'esempio, in alto la soluzione di Galerkin nel primo caso, al centro e in basso un confronto tra la soluzione esatta e quella di Galerkin nel secondo e nel terzo caso. [Hug00]	12
3.1	Esempio di mesh bidimensionali e relative suddivisioni.	16
3.2	Interpolazione mesh bidimensionale.	18
3.3	Differenze di gradi di libertà fra mesh uguali.	19
3.4	Raffigurazione di una simulazione CFD sul profilo aereodinamico di una macchina da corsa.	21
3.5	Raffigurazione di una simulazione CFD per un impianto di riscaldamento.	22
3.6	Raffigurazione di una simulazione CFD della fase iniziale di lancio di un razzo.	23
3.7	Tempesta di sabbia simulata con tecniche CFD, scena tratta dal film 'Mad Max: Fury Road'.	24
3.8	Simulazione di fluidi in tempo reale, oggetto dell'articolo [MM13].	25
4.1	Logo Python	27
4.2	Logo OpenMPI	35
4.3	Logo OpenFoam	42
4.4	Logo Fenics	43
4.5	Architettura deal.II	44
4.6	Logo CMake: CMake è distribuito in modalità open-source da Kitware.	48
4.7	Logo GNU Operating System	50

4.8	Differenze fra macchine virtuali e container Docker	51
4.9	Differenze fra macchine virtuali e container Docker	52
4.10	Logo Docker	53
4.11	Deformazione renderizzata con OSPRay in Paraview	54
4.12	Logo Paraview	55
4.13	Interfaccia SALOME	56
5.1	Interazione fra moduli.	58
5.2	Flusso di gestione input.	60
5.3	diagramma di flusso NavierStokes.	73
5.4	diagramma di flusso Temperatura e laser.	76
5.5	diagramma di flusso Temperatura e laser.	79
5.6	Esempio di mesh distribuita su più thread, differenziati per colore.	85
6.1	Ugello visto dall'alto.	89
6.2	Ugello visto dal basso.	90
6.3	Ugello sezione laterale.	91
6.4	Dettaglio flusso in entrata ugello, sezione laterale. Velocità fluido.	92
6.5	Dettaglio flusso in entrata ugello, sezione laterale. Pressione fluido.	93
6.6	Ugello con radiazione laser, sezione laterale.	94
6.7	Anello di particelle in entrata, vista dall'alto.	96
6.8	Particelle con colorazione basata su valori temperatura, vista laterale.	97
6.9	Dettaglio particelle riscaldate da laser,sezione laterale.	98
6.10	Dettaglio punto di entrata particelle con ugello e flusso, sezione laterale.	99
6.11	Dettaglio punto di entrata particelle con ugello e flusso, sezione laterale.	100

Capitolo 1

Manifattura additiva

Per manifattura additiva si intende un qualsiasi processo di costruzione di oggetti, o parti di oggetti, tridimensionali attraverso il deposito di sottili strati di materia in maniera guidata digitalmente [DWZ⁺18]. Questo tipo di manifattura permette di produrre parti estremamente complesse senza il bisogno di dover costruire costose sagome o stampi per colature a caldo. Un'altro grande vantaggio è quello di poter realizzare parti in singoli blocchi, senza la necessità di dover assemblare pezzi, fattore che implica una rigidità strutturale elevata in quanto eventuali giunti o fori sono assenti. La mancanza di pezzi di assemblaggio rende anche meno costoso lo stoccaggio e la gestione di elementi, abbassando notevolmente il costo per prodotto. I settori che stanno traendo beneficio da questo tipo di manifattura sono quelli aeresopaziali, medici, energetici e automobilistici. Negli ultimi vent'anni la manifattura additiva ha raggiunto livelli di qualità critici, soprattutto grazie all'evoluzione delle piattaforme di high-performance computing, dello stoccaggio e del trattamento dei materiali metallici, dell'efficienza raggiunta dai laser ad alta potenza e dallo studio di leghe metalliche complesse.

Inizialmente la manifattura additiva è stata impiegata soprattutto nel campo della prototipazione di design, componenti la cui funzionalità principale era quella di mostrare un concetto piuttosto che implementarlo. La ricerca nei settori sopracitati ha portato alla sperimentazione e al successivo utilizzo di manifattura additiva con materiali metallici, proprio per l'abilità di creare strutture complesse senza assemblaggio e quindi pronte all'uso. La produzione attraverso manifattura additiva metallica pone

diversi problemi all'attenzione di chi la utilizza, in primis i problemi relativi all'utilizzo in ambienti ad alta temperatura (come per esempio ugelli di motori a reazione).

Il campo della manifattura additiva è relativamente giovane ed è sicuramente ancora agli albori, ma la necessità di creare strutture sempre più complesse, sia in termini di grandezza che in termini di forma, sta spingendo sempre di più la ricerca verso diverse forme e processi, oltre che verso lo studio di innovative leghe metalliche.


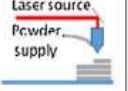
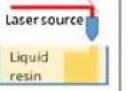


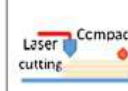
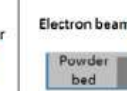
Additive Manufacturing (AM) Processes														
Process	Laser Based AM Processes						Extrusion Thermal	Material Jetting	Material Adhesion	Electron Beam				
	Laser Melting			Laser Polymerization										
Process Schematic														
Name Material	SLS		DMD		SLA		FDM		3DP		LOM		EBM	
	SLM		LENS		SGC		Robocasting		IJP		SFP			
	DMLS		SLC		LTP				MJM					
			LPD		BIS				BPM					
					HIS				Thermojet					
Bulk Material Type														
	Powder		Liquid		Solid									

Figura 1.1: Vari processi di manifattura additiva.

1.1 Processo LMD

Esistono diversi processi per realizzare componenti attraverso la manifattura additiva, questo progetto di tesi prende in evidenza un tipo particolare di processo chiamato Laser Metal Deposition. I processi LMD sono un emergente categoria di processi di manifattura additiva che consistono nella creazione di componenti metalliche dense attraverso l'applicazione di polveri metalliche stratificate, opportunamente fuse attraverso un raggio laser [ZWLS14]. I macchinari che utilizzano questo processo utilizzano un raggio laser che crea un piccolo bacino di polvere fusa o sopra la superficie o sopra uno strato precedentemente fuso. Il laser utilizzato viene regolato attraverso delle lenti focali per concentrarne il fascio alla distanza corretta. Queste polveri sono veicolate sul substrato attraverso un ugello coassiale ebro di gas protettivi anti-ossidazione, per esempio azoto o argon. Il flusso di gas all'interno dell'ugello è fondamentale poiché la scarsità di esso

compromette la posa del materiale in quanto favorisce ossidazione, mentre l'abbondanza aumenta la velocità del flusso e delle particelle spazzandole dalla superficie. La riuscita di questo processo sta proprio nel calibrare correttamente il flusso del gas, la quantità di particelle immesse nell'ugello per unità di tempo e la potenza del laser. Per coadiuvare il processo di posa dei materiali spesso si utilizzano degli ugelli capaci di regolare la distanza dalla superficie, i più avanzati inoltre regolano in maniera indipendente angolo di posa e distanza su più assi. Poiché l'energia cinetica delle particelle in uscita dall'ugello è di gran lunga superiore alla forza di gravità si può anche pensare di depositare materiale con specifici bracci robotici capaci di variare la posizione e l'angolo dell'ugello su 5 o 6 assi, garantendo la costruzione di forme ancora più complesse attraverso la posa verticale dei materiali.

Le particelle solitamente variano in diametro fra $40\mu\text{m}$ e $150\mu\text{m}$. Il materiale usato può variare, vengono usate principalmente leghe d'acciaio, titanio o alluminio.

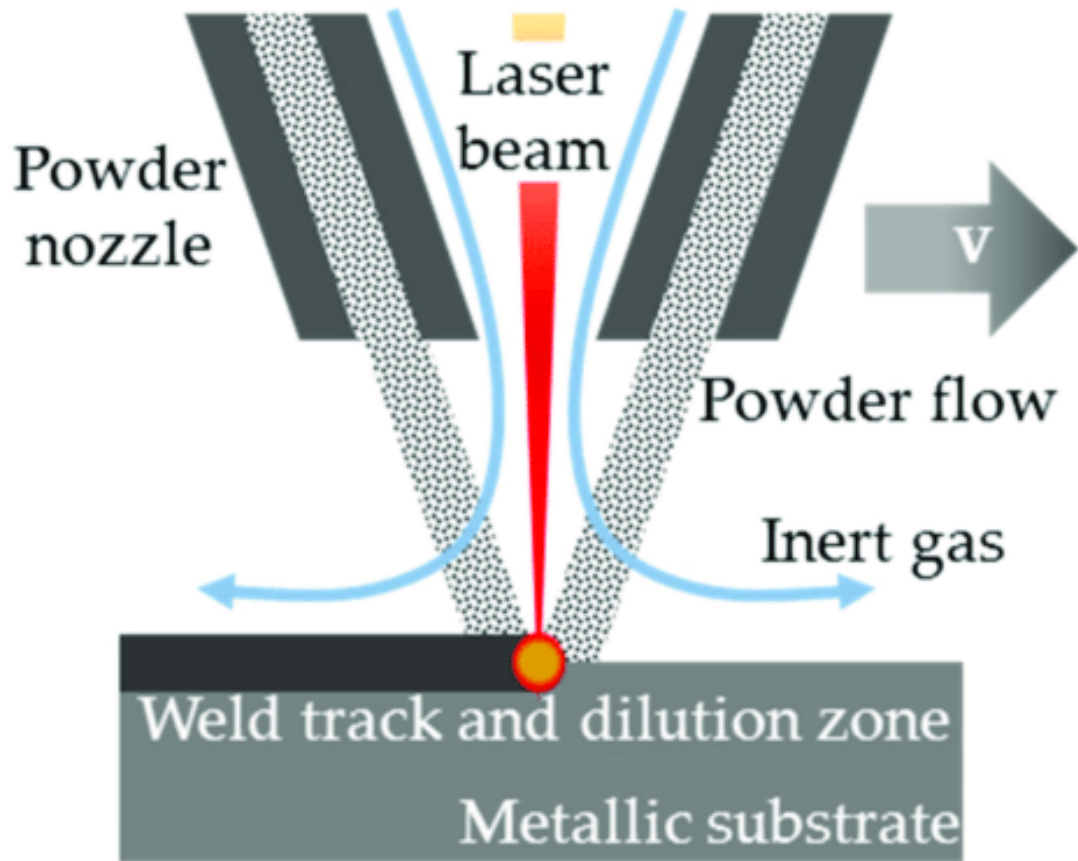


Figura 1.2: Rappresentazione di un ugello LMD.

L'obiettivo principale di questa tesi consiste nel creare un applicativo in grado di simulare numericamente scenari di Laser Metal Deposition da un punto di vista di emissione di particelle. In particolare durante lo sviluppo del solutore sono stati concentrati gli sforzi sul simulare l'interazione tra le particelle di metallo e i seguenti componenti:

- Pareti dei canali dell'ugello (analisi deviazione particelle)
- Flusso di gas
- Temperatura flusso
- Temperatura laser

Capitolo 2

Modello Matematico

In questo capitolo analizzeremo diversi concetti matematici essenziali nella risoluzione numerica di simulazioni CFD e, in particolare, della simulazione oggetto di questa tesi. In particolare ci soffermeremo sulle equazioni utilizzate per simulare correttamente il movimento del gas e delle particelle all'interno dell'ugello.

2.1 Equazioni differenziali alle derivate parziali

In analisi matematica per *Equazione differenziale alle derivate parziali* (PDE in breve) intendiamo una qualsiasi equazione differenziale che coinvolge derivate parziali di una funzione incognita con più variabili indipendenti. Invece di scrivere esplicitamente la funzione essa viene definita indirettamente attraverso una relazione fra se stessa e le sue derivate parziali. La relazione deve necessariamente essere locale, ovvero deve connettere la funzione alle sue derivate nello stesso punto.

Solitamente le PDE vengono utilizzate per risolvere complessi problemi in svariati campi fisici tra cui:

- Elettrostatica
- Elettrodinamica
- Meccanica dei fluidi
- Aereodinamica

- Elasticità
- Meccanica Quantistica
- Relatività

Un'equazione differenziale alle derivate parziali di ordine k ha la seguente forma:

$$F(D^k u(x), D^{k-1} u(x), \dots, Du(x), u(x), x) = 0 \quad (2.1)$$

dove k rappresenta un numero intero, D^k un operatore di derivazione di ordine k rispetto a una o più variabili e infine x appartiene a un sottoinsieme U aperto di \mathbb{R}^n . La funzione F è data, mentre la funzione u è l'incognita dell'equazione. La risoluzione di una PDE consiste nella ricerca di tutte le funzioni u che la rendono un'identità su un opportuno insieme. Di solito viene anche richiesto che la soluzione soddisfi una o più condizioni al contorno ausiliarie. Le PDE possono essere anche non lineari, quando dipende non-linearmente dal più alto grado di derivazione presente.

2.2 Equazioni di Navier-Stokes

In ambito di fluidodinamica le equazioni (PDE) di Navier-Stokes sono un sistema di tre equazioni, dette di bilancio, della meccanica dei corpi continui. Esse descrivono un flusso lineare viscoso e fanno parte dei cosiddetti 7 problemi per il millennio, ovvero problemi la cui difficoltà intrinseca li rende estremamente complessi da risolvere. Difficilmente si possono raggiungere soluzioni analitiche esatte (solo in casi di problemi semplificati), solitamente ci si affida a soluzioni approssimate per i problemi comuni.

2.2.1 Modello matematico

Le equazioni di Navier-Stokes introducono termini come viscosità e conducibilità termica del fluido all'interno delle altrettanto celebri equazioni di bilancio di Eulero, che complicano la soluzione e diminuiscono l'efficienza predittiva delle stesse. Nel caso generale infatti il sistema coinvolge cinque PDE con venti variabili.

Possiamo utilizzare le equazioni di Navier-Stokes per descrivere completamente qualsiasi flusso fluido, anche in presenza di turbolenze. Nel caso di flusso turbolento le

traiettorie di particelle di flusso non sono più costanti nel tempo. Le risorse di calcolo necessarie alla risoluzione di queste traiettorie crescono in base al numero di Reynolds (numero adimensionale usato in fluidodinamica, proporzionale al rapporto tra le forze d'inerzia e le forze viscosi di un fluido) in maniera esponenziale. Le equazioni vengono completate dalle condizioni al contorno e dalle condizioni iniziali. La soluzione a queste equazioni fornisce informazioni sul campo delle velocità del fluido, dalle quali poi si può risalire alle restanti grandezze.

Le equazioni che descrivono il moto di un fluido a densità costante ρ , in un dominio $\Omega \subset \mathbb{R}^d$ (with $d = 2, 3$), sono scritte nella seguente forma:

$$\frac{\partial \mathbf{u}}{\partial t} + (\mathbf{u} \cdot \nabla) \mathbf{u} - \nabla \cdot \tau + \nabla p = \mathbf{f}, \quad \forall \mathbf{x} \in \Omega, t > 0, \quad (2.2)$$

$$\nabla \cdot \mathbf{u} = 0, \quad \forall \mathbf{x} \in \Omega, t > 0, \quad (2.3)$$

dove \mathbf{u} è la velocità del flusso, ρ è la densità del flusso, p è la pressione del flusso divisa per la densità, $\nu = \mu/\rho$ la viscosità cinematica, μ la viscosità dinamica, \mathbf{f} la forza per unità di massa e τ rappresenta lo stress viscoso ed è equivalente a: $\tau = 2\mu\varepsilon$ con $\varepsilon = \frac{1}{2}(\nabla \mathbf{u} + \nabla \mathbf{u}^\top)$. La prima equazione del sistema è l'equazione di bilancio del momento (2.2), la seconda è l'equazione di conservazione della massa (2.3), chiamata anche come equazione di continuità. Il termine $(\mathbf{u} \cdot \nabla) \mathbf{u}$ descrive il processo di trasporto convettivo, mentre $-\nabla \cdot \tau$ descrive il processo di diffusione molecolare. Nel caso in cui la densità sia costante, usando l'equazione di continuità, otteniamo

$$\nabla \cdot \tau = 2\mu \nabla \cdot \varepsilon = \mu \nabla \cdot (\nabla \mathbf{u} + \nabla \mathbf{u}^\top) = \mu \nabla^2 \mathbf{u} \quad (2.4)$$

e il sistema può essere scritto nella forma compatta:

$$\frac{\partial \mathbf{u}}{\partial t} + (\mathbf{u} \cdot \nabla) \mathbf{u} - \nu \Delta \mathbf{u} + \nabla p = \mathbf{f}, \quad \forall \mathbf{x} \in \Omega, t > 0, \quad (2.5)$$

$$\nabla \cdot \mathbf{u} = 0, \quad \forall \mathbf{x} \in \Omega, t > 0. \quad (2.6)$$

Le equazioni (2.5) e (2.6) sono chiamate equazioni incompressibili di Navier-Stokes. Generalmente, i flussi che soddisfano queste condizioni di incompressibilità $\nabla \cdot \mathbf{u} = 0$, sono chiamati fluidi incompressibili. Come anticipato è necessario aggiungere delle condizioni iniziali per impostare correttamente il problema, tali sono:

$$\mathbf{u}(\mathbf{x}, 0) = \mathbf{u}_0(\mathbf{x}) \quad \forall \mathbf{x} \in \Omega \quad (2.7)$$

con le relative condizioni al contorno:

$$\mathbf{u}(\mathbf{x}, t) = \mathbf{u}_D(\mathbf{x}, t) \quad \forall \mathbf{x} \in \Gamma_D, \quad (2.8)$$

$$\left(\nu \frac{\partial \mathbf{u}}{\partial \mathbf{n}} - p \mathbf{n} \right)(\mathbf{x}, t) = \mathbf{t}(\mathbf{x}, t) \quad \forall \mathbf{x} \in \Gamma_N, \quad (2.9)$$

Le equazioni (2.5) e (2.6) sono generalmente più semplici da risolvere da un punto di vista computazionale, sono quindi quelle su cui si basa il modello matematico del simulatore oggetto della tesi.

2.3 Metodo elementi finiti

Il metodo degli elementi finiti, o **FEM** dall'inglese *finite element method*, è una tecnica numerica che cerca soluzioni approssimate a problemi descritti da PDE (). Compete con altri metodi ben noti quali: metodo delle differenze finite, metodo dei volumi finiti, metodo degli elementi al contorno, metodo delle celle, metodo spettrale ma rimane tuttora il più utilizzato in ambito computazionale. Questo metodo si presta soprattutto a problemi dettati da PDE aventi domini di forma complessa o variabile, quando il livello di accuratezza richiesto non è omogeneo sul dominio (si veda in 3.2.1 l'accuratezza in prossimità dell'ugello) e quando la soluzione cercata pecca di regolarità. Il metodo FEM è anche utilizzato per discretizzare il problema dei flussi incomprimibili visto nella precedente sezione, ed è alla base della soluzione implementata in questa tesi.

In generale questi metodi prevedono la discretizzazione di un dominio in griglie o mesh di punti in cui computare soluzioni basate su discreti step temporali. Su ciascun elemento del dominio discretizzato viene espressa una combinazione lineare di funzioni dette funzioni forma, o shape functions, che spesso vengono approssimate. L'accuratezza delle soluzioni trovate per queste funzioni è legata al grado del polinomio che le compone.

Il metodo degli elementi finiti fa parte della classe del metodo di Galërkin, metodi di analisi numerica che permettono di passare da uno spazio continuo a uno discreto, il cui punto di partenza è la cosiddetta formulazione debole di un problema differenziale. La formulazione debole è essenziale poiché richiede alla soluzione di soddisfare determinati requisiti di regolarità, che sono realistiche per quasi tutti i problemi d'ingegneria a cui si applicano. I metodi di tipo Galërkin approssimano la soluzione del problema scritto

in forma debole mediante la combinazione lineare delle funzioni di forma elementari. I coefficienti di queste combinazioni lineari sono i gradi di libertà del problema che rappresentano le incognite del problema ottenuto dalla discretizzazione.

2.3.1 Esempio FEM con un grado di libertà

Proponiamo di seguito un esempio di problema risolvibile con FEM [Hug00]: $u_{,xx} + f = 0$ la virgola indica la differenziazione mentre f è una funzione continua e derivabile nell'intervallo $[0, 1]$. Un problema ai limiti richiede l'imposizione di alcune condizioni al contorno sulla funzione u , supponiamo quindi:

$$u(1) = q \text{ e } -u_{,x}(0) = h \quad (2.10)$$

Con q e h costanti. La formula forte del problema sarà dunque la seguente:

$$(S) \left\{ \begin{array}{l} \text{Dati } f : \bar{\Omega} \rightarrow \mathbb{R} \text{ e le costanti } q \text{ e } h, \text{ trovare } u : \bar{\Omega} \rightarrow \mathbb{R}, \text{ tale che} \\ u_{,xx} + f = 0 \\ u(1) = q \\ -u_{,x}(0) = h \end{array} \right. \quad (2.11)$$

Dove Ω appartiene all'intervallo aperto tra 0 e 1, mentre $\bar{\Omega}$ al medesimo chiuso. La soluzione esatta è:

$$u(x) = q + (1-x)h + \int_x^1 \left\{ \int_0^y f(z)dz \right\} dy \quad (2.12)$$

dove con y e z indichiamo delle variabili di comodo. Per definire la forma debole di S definiamo due classi di funzioni:

- *Soluzioni di prova*, indicate con S , tali per cui:

$$S = \{ u \mid u \in H^1, u(1) = q \} \quad (2.13)$$

- *Funzioni peso*, indicate con \mathcal{V} , tali per cui:

$$\mathcal{V} = \{ w \mid w \in H^1, w(1) = 0 \} \quad (2.14)$$

Ciò detto, è possibile definire la formulazione debole:

$$(W) \left\{ \begin{array}{l} \text{Dati } f, q, \text{ e } h \text{ come prima, trovare } u \in \mathcal{S}, \text{ tale che per ogni } w \in \mathcal{V} \\ \int_0^1 w_{,x} u_{,x} dx = \int_0^1 w f dx + w(0)h \end{array} \right. \quad (2.15)$$

Le formulazioni, debole e forte, sono equivalenti, perciò per ottenere una soluzione valida, pur sempre approssimata, il metodo degli elementi finiti considera la formulazione debole. Come citato in precedenza un modo per ottenere tale soluzione è attraverso il *metodo di Galerkin*. In primis procediamo alla costruzione delle approssimazioni di dimensione finita $\mathcal{S}^h \subset \mathcal{S}$ e $\mathcal{V}^h \subset \mathcal{V}$. Quindi per ogni $v^h \in \mathcal{V}^h$ costruiamo una funzione $u^h \in \mathcal{S}^h$:

$$a(w^h, u^h) = (w^h, f) + w^h(0)h \quad (2.16)$$

Quest'ultima definisce una soluzione approssimata u^h . Sostituendo la precedente, definiamo la forma di Galerkin del problema:

$$(G) \left\{ \begin{array}{l} \text{Dati } f, q, \text{ e } h \text{ come prima, trovare } u^h = v^h + q^h, \text{ dove } v^h \in \mathcal{V}^h \\ \text{tale che per ogni } w^h \in \mathcal{V}^h \\ a(w^h, v^h) = (w^h, f) + w^h(0)h - a(w^h, q^h) \end{array} \right. \quad (2.17)$$

Riscrivibile come:

$$\sum_{B=1}^n a(N_A, N_B) d_B = (N_A, f) + N_A(0)h - a(N_A, N_{n+1}q) \quad (2.18)$$

Dove N_A, N_B e N_{n+1} sono dette funzioni di forma, per le quali $N_A(1) = 0$, $N_B(1) = 0$, $N_{n+1}(1) = 1$. Ciò rappresenta un sistema di n equazioni in n incognite, che possiamo riscrivere come:

$$\sum_{B=1}^n K_{AB} d_B = F_A, \quad A = 1, 2, \dots, n \quad (2.19)$$

Dove:

$$\begin{aligned} K_{AB} &= a(N_A, N_B) \\ F_A &= (N_A, f) + N_A(0)h - a(N_A, N_{n+1}q) \end{aligned} \quad (2.20)$$

Definiamo ora le matrici:

$$K = [K_{AB}] = \begin{bmatrix} K_{11} & \dots & K_{1n} \\ \vdots & & \vdots \\ K_{n1} & \dots & K_{nn} \end{bmatrix} \quad (2.21)$$

$$F = \{F_A\} = \begin{Bmatrix} F_1 \\ \vdots \\ F_n \end{Bmatrix} \quad (2.22)$$

$$d = \{d_B\} = \begin{Bmatrix} d_1 \\ \vdots \\ d_n \end{Bmatrix} \quad (2.23)$$

Riformulando il problema di Galerkin in:

$$(M) \begin{cases} \text{Date la matrice di coefficienti } K \text{ e il vettore } F, \text{ trovare } d \text{ tale che} \\ Kd = F \end{cases} \quad (2.24)$$

Per un'applicazione 1D: si prenda $n = 1$, allora $w^h = c_1 N_1$, $eu^h = v^h + q^h = d_1 N_1 + q N_2$. L'unica incognita è d_1 . Le funzioni di forma devono soddisfare $N_1(1) = 0$ e $N_2(1) = 0$. Prendiamo $N_1(x) = 1 - x$ e $N_2(x)$. Le matrici introdotte in precedenza diventano quindi:

$$\begin{aligned} K &= \{K_{11}\} = K_{11} = a(N_1, N_1) = \int_0^1 N_{1,x} N_{1,x} dx = 1 \\ F &= \{F_1\} = F_1 = (F_1, f) + N_1(0)h - a(N_1, N_2)q \\ &= \int_0^1 (1-x)f(x)dx + h - \int_0^1 N_{1,x} N_{2,x} dx q = \int_0^1 (1-x)f(x)dx + h + q \\ d &= \{d_1\} = d_1 = K_{11}^{-1} F_1 = F_1 \end{aligned} \quad (2.25)$$

Di conseguenza:

$$u^h(x) = d_1(1-x) + qx \quad (2.26)$$

Se $f = 0$:

$$u^h(x) = u(x) = q + (1-x)h \quad (2.27)$$

In questo caso la soluzione approssimata è esatta. Se f è uguale a una costante $f(x) = p$:

$$\begin{aligned} u_{\text{part}}(x) &= \frac{p(1-x^2)}{2} \\ u_{\text{part}}^h(x) &= \frac{p(1-x)}{2} \end{aligned} \quad (2.28)$$

Se invece $f(x) = qx$, dove q è una costante, allora avremo:

$$\begin{aligned} u_{\text{part}}(x) &= \frac{q(1-x^3)}{6} \\ u_{\text{part}}^h(x) &= \frac{q(1-x)}{6} \end{aligned} \quad (2.29)$$

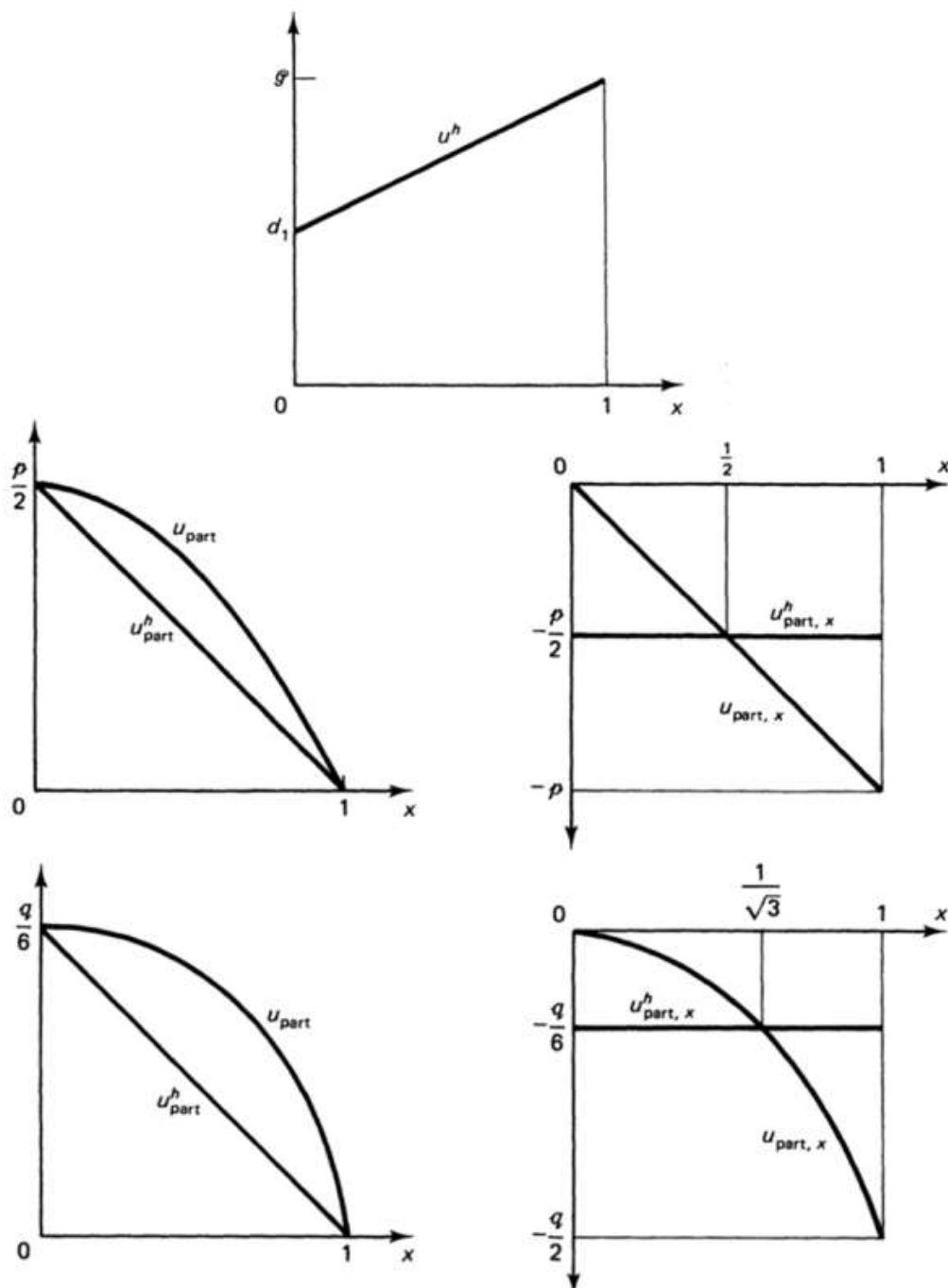


Figura 2.1: Soluzioni dell'esempio, in alto la soluzione di Galerkin nel primo caso, al centro e in basso un confronto tra la soluzione esatta e quella di Galerkin nel secondo e nel terzo caso. [Hug00]

2.4 Metodo Newton-Raphson

Il metodo Newton-Raphson, detto anche metodo delle tangenti, è uno dei metodi utilizzati per calcolare approssimativamente la soluzione di un'equazione nella forma:

$$f(x) = 0 \quad (2.30)$$

Il metodo consiste nel sostituire a una generica curva $y = f(x)$ la tangente alla curva stessa, partendo da un qualsiasi punto. La relazione di ricorrenza del metodo è la seguente:

$$x_{n+1} = x_n - \frac{f(x_n)}{df(x_n)} \quad (2.31)$$

dove n rappresenta il numero di approssimazioni. Questo metodo è stato utilizzato principalmente per approssimare le soluzioni delle funzioni di forma, nel modulo di Navier-Stokes (5.4).

Capitolo 3

Computazione Fluidodinamica

La fluidodinamica computazionale, spesso abbreviata con CFD, è un metodo per risolvere e analizzare problemi di fluidodinamica attraverso l'analisi numerica su calcolatore. Viene utilizzata specialmente nel campo dell'industria e della ricerca per tutto ciò che concerne problematiche che riguardano l'interazioni di fluidi con altre superfici all'interno di un dominio. Alla base del funzionamento di tutti i software di CFD risiede la capacità di risolvere le equazioni di Navier-Stokes (2.2) congiuntamente alle equazioni ad esse collegate. Come anticipato nel Capitolo 2 la soluzione esatta a questi problemi accade solo in presenza di semplici flussi laminari che interagiscono con geometrie semplici, la maggior parte dei casi ovviamente non ricade in questi casi e risultano quindi necessari degli approcci approssimati attraverso metodi numerici.

Analizziamo di seguito i passi necessari per portare a termine una generica simulazione fluidodinamica, definiti in base all'ordine con cui vengono eseguiti solitamente.

3.1 Modello 3D

Un modello tridimensionale, o 3D, è la rappresentazione di un oggetto tridimensionale basata su notazione matematica. Risiede alla base di ogni simulazione CFD poiché rappresenta l'oggetto con cui il fluido da simulare interagirà. A seconda delle necessità legate alla simulazione è possibile realizzare modelli di superfici ideali (superfici in tre dimensioni ma con spessore infinitamente piccolo), anche chiamati Boundary-based objects, o di oggetti solidi che occupano un volume, detti Volume-based objects, spesso

costruiti attraverso forme geometriche lementari. Gli oggetti basati su volumi sono quelli più usati in ambito CFD perché rappresentano un vero e proprio corpo con proprietà fisiche reali (massa, volume, centro di gravità, momenti, densità). Proprio grazie a queste proprietà possiamo influenzare le simulazioni CFD. Il modello 3D di ciò che si sta simulando può essere considerato come il "negativo" della mesh, ovvero l'oggetto da cui viene discretizzato il volume e creata la relativa mesh.

3.2 Discretizzazione del volume

Uno dei processi più delicati delle simulazioni CFD è proprio quello della discretizzazione del dominio di interesse. Discretizzare significa trasformare modelli matematici ed equazioni continue in entità discrete, ed è alla base di ogni tipo di computazione matematica effettuata da un calcolatore. Nel caso della discretizzazione di un volume si tenta di trasformare una regione di spazio continuo tridimensionale in un oggetto comprensibile dal calcolatore, spesso si tratta di un oggetto composto da punti dotati di tre dimensioni che formano delle geometrie tridimensionali. Intrinsecamente a ogni discretizzazione vi è una inevitabile perdita d'informazione, minore questa perdita e maggiore sarà la risoluzione della simulazione.

3.2.1 Mesh

Il processo che discretizza un dominio è chiamato anche generazione di mesh che non è altro che l'insieme di tutti gli elementi utilizzati per discretizzare lo stesso. In termini pratici una mesh è un reticolo di punti che definiscono un oggetto, composto da vertici, spigoli e facce. La mesh è composta da geometrie semplici primitive definite matematicamente (detti anche elementi finiti), principalmente si tratta di triangoli o quadrilateri per i domini bidimensionali e di esaedri e tetraedri per quelli tridimensionali. Naturalmente per modellare particolari curve, o in generale elementi complessi, è possibile avvalersi di diverse geometrie primitive. Una volta creata una mesh è possibile aumentarne la risoluzione attraverso processi di suddivisione che rendono più accurati i risultati generati, incrementando significativamente i tempi di simulazione.

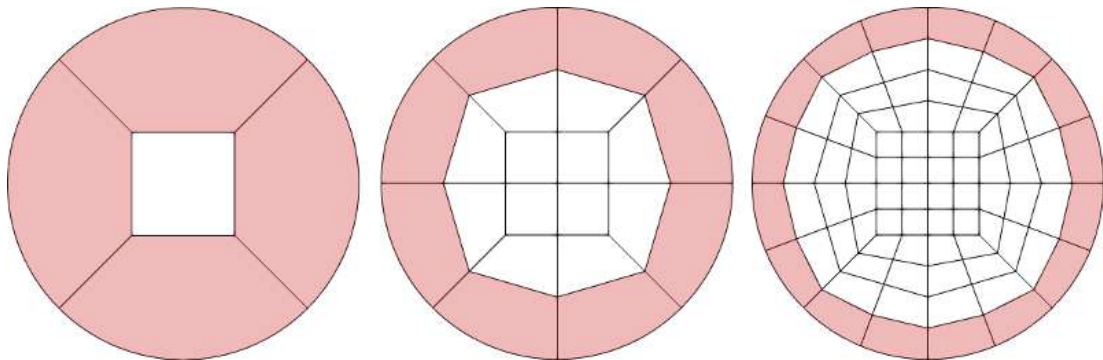


Figura 3.1: Esempio di mesh bidimensionali e relative suddivisioni.

3.3 Solver

Nei capitoli precedenti abbiamo descritto i problemi la cui risoluzione è necessaria per computare correttamente le simulazioni fluidodinamiche, in questa sezione analizziamo come agiscono i solutori (Solver) per trovare correttamente e nel minor tempo possibile tali soluzioni. Un solver non è altro che un qualsiasi programma, o parte di programma, che dati in input:

- Mesh
- Equazioni Navier-Stokes
- Condizioni al contorno
- Costanti fisiche
- Parametri di discretizzazione temporale

riesce a computare una soluzione accettabile e produrre in output una serie di risultati comprensibili da programmi di visualizzazione o post processing. Nelle sottosezioni seguenti analizzeremo nel dettaglio alcuni importanti concetti relativi al solutore argomento di questa tesi.

3.3.1 Condizioni al contorno

Le condizioni iniziali e al contorno sono necessarie a definire correttamente il problema che si sta cercando di risolvere. Rappresentano parametri di cui conosciamo il comportamento durante il tempo e che possiamo inserire all'interno delle nostre equazioni di Navier-Stokes per tentare di risolvere il sistema. Esempi di condizione al contorno, contestualizzati con il caso in esame, possono essere la velocità di ingresso del flusso all'interno dell'ugello, la temperatura iniziale del flusso, la pressione iniziale del flusso, il volume irradiato dal fascio laser. Altri tipi di condizioni al contorno molto importanti per lo sviluppo di questo progetto sono tutte la definizione di tutte le superfici che si comportano come ostacolo per il flusso e le particelle, comunemente chiamati wall (muri). I wall di solito vengono definiti per mezzo di ulteriori parametri quali: rugosità, spessore, temperatura, ecc...

3.3.2 Interpolazione

Per interpolazione si intende il processo atto a individuare, a partire da un insieme finito di punti, nuovi punti all'interno di un sistema di coordinate attraverso delle funzioni. Nelle simulazioni CFD la fase di interpolazione è critica in quanto è quella che effettivamente calcola, per ogni grado di libertà della simulazione, la soluzione delle funzioni associate calcolate nel determinato punto. La figura 3.2 rappresenta un esempio di interpolazione su mesh di punti tridimensionali, notare lo spostamento degli stessi tra il frame iniziale, indicato tramite *wireframe* e quello finale . Ogni punto è stato spostato in accordo con delle funzioni di forma precedentemente impostate su ogni nodo della mesh.

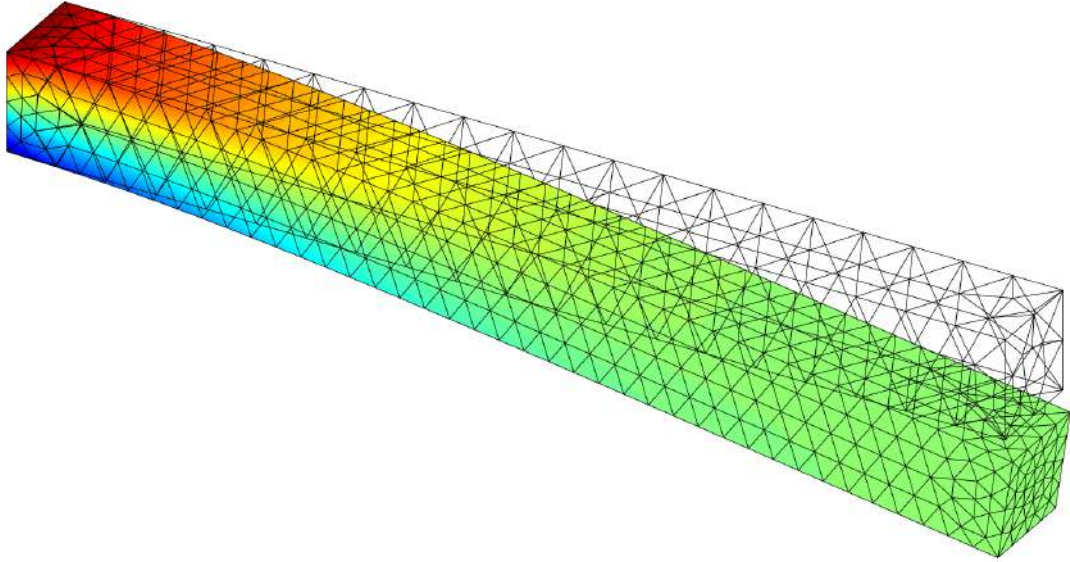


Figura 3.2: Interpolazione mesh bidimensionale.

3.3.3 Gradi di libertà

I gradi di libertà rappresentano il numero di componenti, in termini di variabili, da identificare per determinare completamente la soluzione di un problema. Ogni nodo della mesh con cui si sta lavorando può avere uno o più gradi di libertà. Per esempio il nodo di una mesh che modella un fluido può avere i seguenti gradi di libertà (componenti che variano):

- Tre componenti per la traslazione
- Pressione
- Temperatura

per un totale di cinque gradi di libertà per nodo. In una mesh con 100 nodi ci troveremo a dover calcolare la variazione di 500 gradi di libertà per ogni time step. Essendo parametri che governano lo stato fisico di un sistema possono variare in base alle circostanze, ma in ogni caso queste dimensioni vanno regolate da funzioni altrimenti non sarebbe possibile computare soluzioni per esse. Nella figura 3.3 è possibile notare come per una stessa

superficie si possono impostare diversi livelli di risoluzione attraverso l'aumento di nodi e, quindi, gradi di libertà.

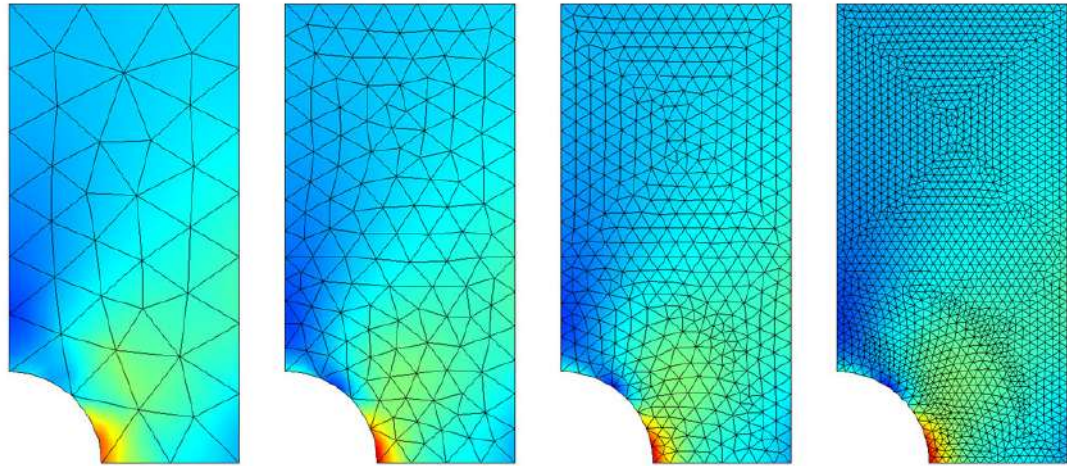


Figura 3.3: Differenze di gradi di libertà fra mesh uguali.

3.3.4 Metodi Lagrangiani-Euleriani

Il solver necessita dei metodi numerici e delle particolari istruzioni per interpretare nel modo corretto il problema e trovare una soluzione, in questa sottosezione analizzeremo l'approccio utilizzato dal solver implementato durante lo sviluppo di questo progetto.

Tipicamente i metodi per predire il movimento di un flusso di particelle si rifanno all'approccio Lagrangiano, particolarmente adatto a tracciare il movimento delle stesse, mentre per Flussi (non di particelle) l'approccio Euleriano va per la maggiore. L'approccio utilizzato per implementare il solver in questione è un particolare metodo basato su Eulero nel calcolo del flusso del gas presente nell'ugello e su Lagrange per quanto riguarda il flusso di particelle metalliche. Si tratta quindi di un approccio ibrido che combina entrambi i metodi per computare più efficacemente la soluzione. Analizziamo nel dettaglio in cosa differiscono i due metodi nelle prossime due sottosezioni.

3.3.4.1 Metodo Lagrangiano

Il metodo Lagrangiano è caratterizzato poiché si studiano gli elementi in base al cambiamento della loro traiettoria. L'approccio Lagrangiano discretizza la massa delle par-

ticelle e quindi simula il fluido attraverso il movimento delle particelle stesse del fluido. Maggiore è il numero di particelle maggiore sarà la risoluzione della simulazione computata. Il metodo Lagrangiano, all'interno di questo progetto, è stato implementato seguendo proprio il concetto di tracciamento di particella; quindi ogni particella all'interno della simulazione viene aggiornata in tutte le sue proprietà secondo l'influenza del flusso (calcolato in modo Euleriano) per ogni time step. Nella pratica questo meccanismo si potrebbe tradurre nel monitoraggio delle proprietà e della posizione di una singola particella metallica durante tutto il percorso all'interno, e all'esterno prima della posa, dell'ugello.

3.3.4.2 Metodo Euleriano

Nel caso del metodo Euleriano è solido trattare direttamente i volumi di fluido fissi nello spazio, studiandone la variazione nello spazio e nel tempo. Viene infatti discretizzato lo spazio del dominio introducendo una griglia fissa (3.2.1) i cui nodi vengono definiti con dei valori che descrivono l'evoluzione del fluido nel tempo. Questi valori che descrivono i nodi della griglia non sono altro che i gradi di libertà accennati prima (3.3.3). Uno dei problemi che si pongono utilizzando questo metodo è il modo in cui si fissa la risoluzione della simulazione in base alla griglia, che ricordiamo essere fissa. In parte si può ovviare a questo problema adoperando dei metodi Euleriani adattivi. Volendo fornire un esempio pratico anche per questo metodo, possiamo visualizzare la tecnica Euleriana come il monitoraggio delle proprietà del flusso in determinati punti fissi del dominio attorno all'ugello.

3.3.5 Output

Una volta calcolate le soluzioni ai problemi precedentemente descritti è necessario fornire i risultati in formati comprensibili da programmi di visualizzazione o post processing (4.4). Spesso i solutori tendono a produrre gruppi di file, almeno uno per time step, per garantire la visualizzazione dell'evoluzione della simulazione durante il corso del tempo, opportunamente discretizzato. Per ottenere la massima flessibilità di visualizzazione e post processing dopo la risoluzione della simulazione è necessario includere quante più informazioni possibili all'interno dei file prodotti. In particolare è opportuno salvare ogni

valore, di ogni proprietà dei nodi e delle particelle, per ogni time step così da rendere possibili avanzate tecniche di analisi di andamenti scalari (semplici grafici valore-tempo) e vettoriali (tracing).

3.4 Applicazioni comuni

Forti di quanto appreso fino a questo punto possiamo analizzare alcune, ma di certo non tutte, applicazioni comuni che tutt'ora sono utilizzate in ambito CFD. Sicuramente in ambito ingegneristico le tecniche simulative CFD trovano riscontro in molte realtà, tra queste vale la pena citarne alcune:

- **Analisi flussi turbolenti e cavitazioni:** la CFD è alla base dello studio riguardo la riduzione (in casi rari dell'aumento) delle turbolenze indotte da superfici. Basti pensare alla progettazione di autoveicoli da corsa il cui profilo aereodinamico è alla base delle alte prestazioni in situazioni ad alta velocità. Nella progettazione di elementi rotativi immersi in fluidi (eliche marine o non) si cerca di ridurre al minimo turbolenze e cavitazioni.

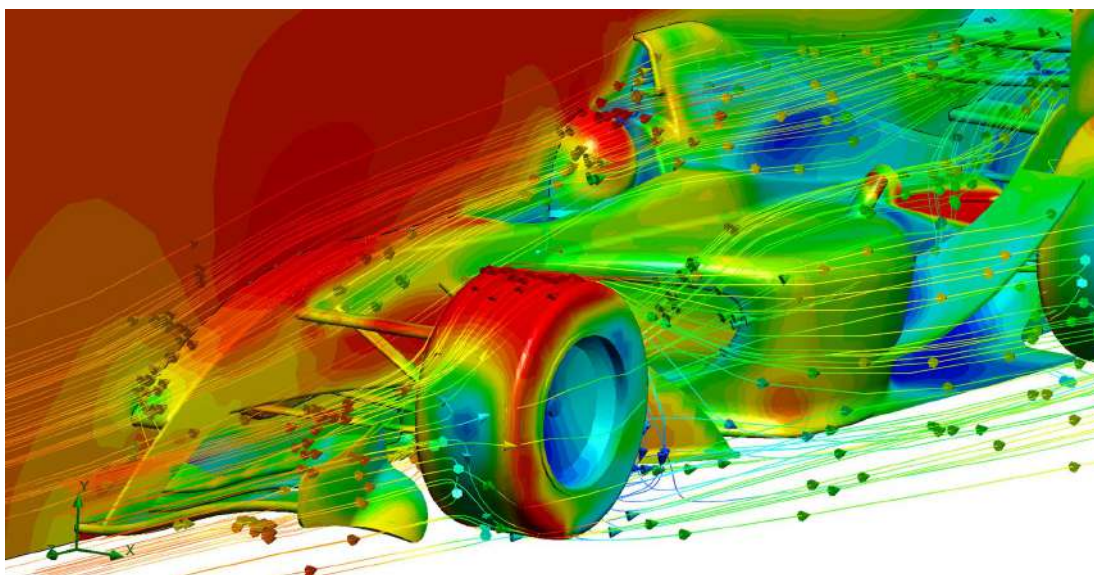


Figura 3.4: Rappresentazione di una simulazione CFD sul profilo aereodinamico di una macchina da corsa.

- **Analisi termica e barometrica di flussi:** in questa categoria ricadono tutte quelle simulazioni atte a predire il comportamento di fluidi caldi o freddi all'interno di strutture solide soggette a espansioni termiche. Un esempio può essere la simulazione di complessi impianti idraulici e di riscaldamento centralizzati.

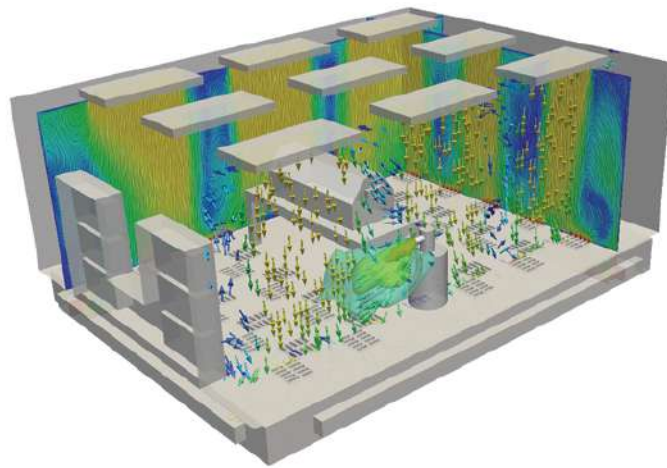


Figura 3.5: Rappresentazione di una simulazione CFD per un impianto di riscaldamento.

- **Analisi di performance motori a stato solido/liquido per razzi e velivoli:** un'analisi dettagliata di come il flusso di un motore a reazione vari in base a determinate proprietà del fluido è di vitale importanza nella progettazione dello stesso. La CFD taglia notevolmente i costi di implementazione di nuove tecnologie in quanto è possibile ottenere risultati particolarmente precisi senza dover affrontare l'intero processo produttivo, ma utilizzando esclusivamente risultati numerici propriamente verificati.

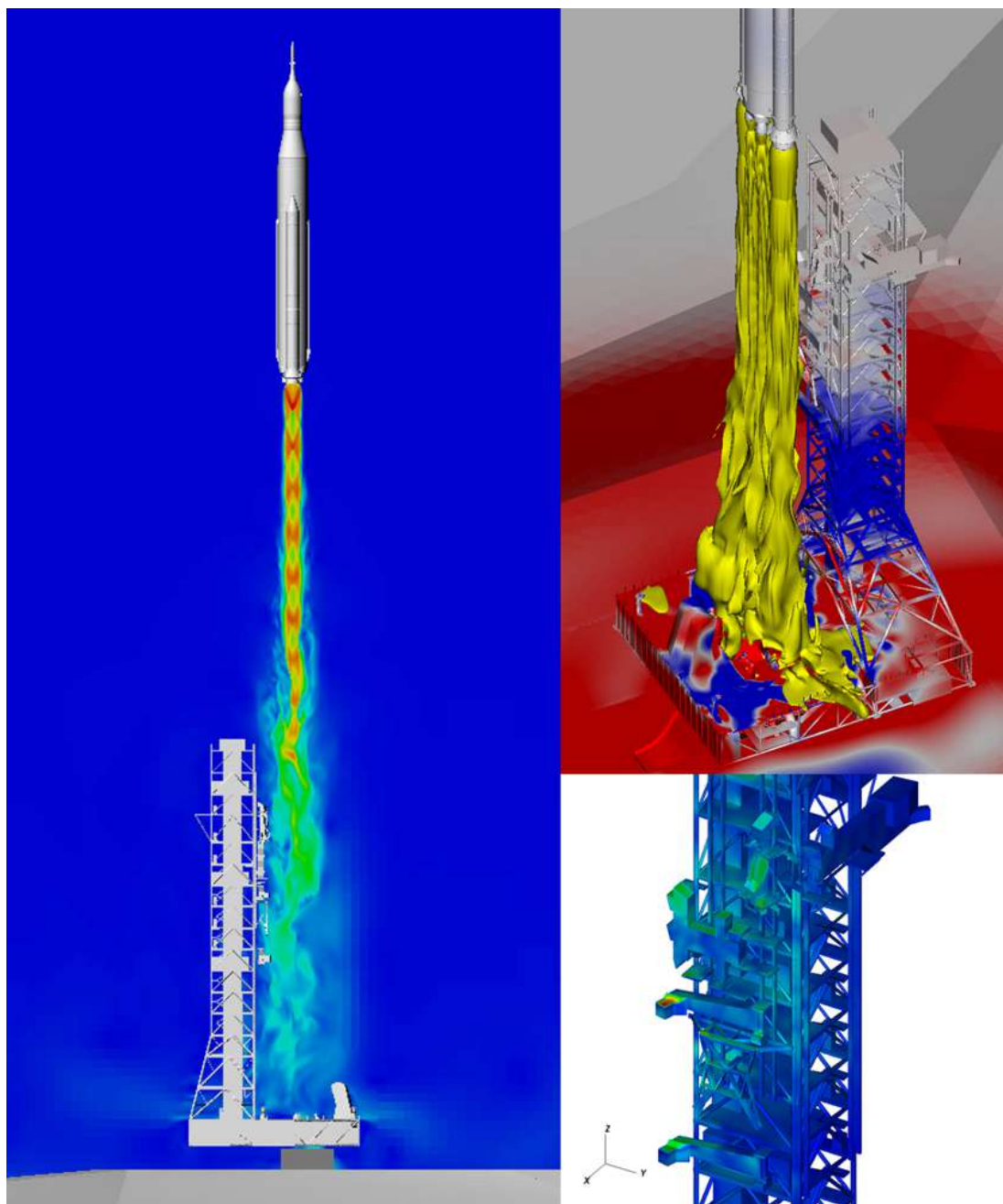


Figura 3.6: Rappresentazione di una simulazione CFD della fase iniziale di lancio di un razzo.

La CFD, da qualche anno a questa parte, ha fatto il suo debutto anche in ambiti d'intrattenimento. Le basi della CFD infatti sono state adattate a problemi relativamente

più semplici, che non richiedono una soluzione numericamente precisa, bensì più mirata a produrre simulazioni visivamente convincenti. Questi ambiti riguardano principalmente l'industria cinematografica e videoludica. Queste ultime in particolare stanno spingendo molto la ricerca di soluzioni CFD estremamente veloci e parallelizzabili su schede video per simulare al meglio effetti visivi convincenti.



Figura 3.7: Tempesta di sabbia simulata con tecniche CFD, scena tratta dal film 'Mad Max: Fury Road'.

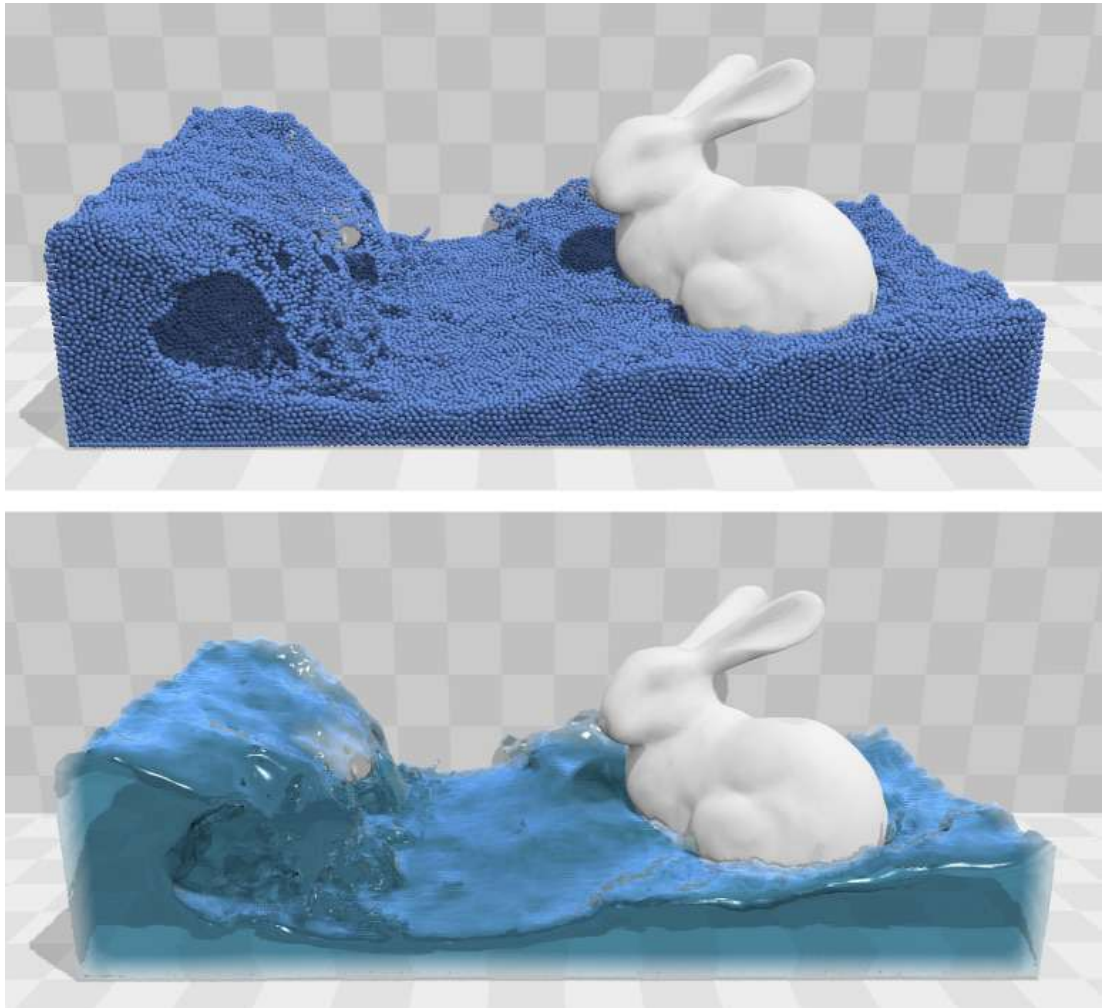


Figura 3.8: Simulazione di fluidi in tempo reale, oggetto dell'articolo [MM13].

Capitolo 4

Background Tecnologico

Come accennato in precedenza, le simulazioni fluidodinamiche risultano particolarmente onerose da un punto di vista computazionale, sia in termini di hardware che di software. Da un punto di vista hardware la numerosa quantità di particelle, celle, volumi e mesh ad alta risoluzione possono gravare sulla memoria primaria del calcolatore. D'altro canto la quantità di questi elementi inficia negativamente anche le prestazioni del processore che viene oberato dei calcoli matematici descritti nel capitolo 2.

Da un punto di vista software invece le difficoltà si celano nel tentativo di ottimizzazione dell'utilizzo delle risorse computazionali. Proprio per questo motivo le scelte di linguaggio, compilatore e librerie risultano critiche da un punto di vista prestazionale.

4.1 Linguaggi Utilizzati

Inizialmente il simulatore venne abbozzato utilizzando Python come linguaggio e FEnics [ABH⁺15] come risolutore di equazioni differenziali alle derivate parziali e LEOPart [MRS19] per simulare l'avvezione delle particelle. Per quanto sia apprezzabile la facilità d'uso di Python si è rivelato in poco tempo un linguaggio poco adatto al nostro caso d'uso (la spiegazione nella sottosezione dedicata), si è optato quindi per utilizzare un linguaggio più performante come C++ e che, soprattutto, è molto più usato per simulazioni di questo tipo. La popolarità di C++ nel campo delle simulazioni, in generale laddove sono necessarie performance (videogiochi 3D, High Performance Computing,

sistemi embedded, industria cinematografica), rende anche più facile reperire materiale, librerie e pubblicazioni riguardo l'argomento.

4.1.1 Python versione 3.6

Python è un linguaggio di programmazione ad alto livello, orientato a oggetti, molto utilizzato in ambito matematico, fisico e scientifico ideato da Guido van Rossum[VRD09]. L'obiettivo principale di Python è dare la possibilità agli utenti di realizzare complesse applicazioni in modo semplice e veloce. La sintassi di Python obiettivamente risulta semplice combinata con la possibilità di utilizzare il paradigma di programmazione orientata agli oggetti lo rendono estremamente leggibile e manutenibile. Il fatto che Python sia un linguaggio interpretato rende anche veloce il ciclo di sviluppo (modifica, test, rilascio). La possibilità di estendere Python con librerie C, C++ è ciò che lo rende utilizzabile anche per computazioni complesse. L'estendibilità è un punto di forza di Python, esistono infatti una grande varietà di librerie di terze parti per soddisfare pressoché qualsiasi caso d'uso. Python viene sviluppato con licenza OpenSource.



Figura 4.1: Logo Python

4.1.2 C++

C++ è un linguaggio di programmazione compilato, di livello più basso rispetto a Python, inventato da Bjarne Stroustrup come successore del linguaggio C e, dal 1998, standardizzato da WG21 [ISO98]. Nel corso degli anni sono state rilasciate diverse

versioni con migliorie e nuove funzionalità, tra le più recenti ci sono: C++17 [ISO17] e C++20 [Dus19]. Il linguaggio si presta a diversi paradigmi di programmazione tra cui:

- Orientata a oggetti
- Funzionale
- Procedurale

Alcune caratteristiche rilevanti del linguaggio:

- Tipizzazione forte: assicura che eventuali errori di compilazione siano rilevati subito
- Insicurezza di esecuzione: gli errori non sono completamente gestiti e si ha accesso diretto alla memoria dei processi
- Multi-piattaforma
- Gestione della memoria manuale

Come anticipato C++ fa affidamento a un compilatore per generare eseguibili, il processo di compilazione può essere suddiviso in tre principali passi:

Preprocessamento: Il preprocessore è responsabile di gestire tutte le direttive di preprocessamento tra cui:

- `#include` Per referenziare librerie esterne o interne all'interno di un file.
- `#define MACRO=1` Per definire un simbolo, o MACRO, che verrà poi sostituito all'interno del codice.
- `#if #ifdef #ifndef` Per abilitare o disabilitare porzioni di codice.

Dopo aver applicato le corrette sostituzioni alle varie direttive il preprocessore produce un artefatto consumabile dal compilatore.

Compilazione: Gli artefatti che arrivano dal preprocessore vengono trattati dal compilatore come puro codice sorgente C++, che li interpreta e li converte in codice macchina producendo diversi file binari (uno per file di codice). Questa fase è responsabile per la generazione di librerie statiche, utili per il riuso del codice. In questa fase vengono sollevati gli errori di compilazione.

Linking: Il "linker" è il componente che produce eseguibili, librerie condivise (o dinamiche) collegando i diversi artefatti prodotti dal compilatore. Il processo prevede la referenziazione di tutti i simboli lasciati indefiniti dal compilatore con gli oggetti corretti, sia che essi siano in librerie condivise o in altri file.

La grande varietà di compilatori disponibili rende C++ portabile e multi-piattaforma, inoltre essendo un'evoluzione di C è anche compatibile con codice C e le librerie sviluppate per esso.

4.1.2.1 Template Metaprogramming

Deal.II 4.2.5 fa largo uso di un sistema di metaprogrammazione presente in C++. Il sistema permette di eseguire svariate operazioni in fase di compilazione che elaborano il codice sorgente. Prevalentemente le operazioni vengono svolte sui tipi utilizzati all'interno del programma. In C++ è presente un rigoroso controllo statico dei tipi che rende necessario definire a priori il tipo delle variabili che vengono usate come argomenti per funzioni o membri di classe. Questo fattore va contro uno dei principi fondamentali della programmazione, il riutilizzo del codice. Per ovviare a questo vincolo C++ offre una parola chiave che viene usata per indicare porzioni di codice generiche, *template*. Esempio di funzione che somma due tipi generici:

```
template <typename T>
T somma_tra_generics(T a, T b) {
    return a+b;
};
int main() {
    int i1 = 1;
```



```
    int i2 = 2;
    int sum = somma_tra_generics(i1,i2);
    std::cout << sum << std::endl;
    return 0;
}
```

In fase di pre-compilazione ogni funzione, o classe, che viene preceduta dalla parola chiave *template* viene processata generando a sua volta nuove funzioni e nuove classi per ogni tipo di variabile con cui essa viene convocata. Al contrario di linguaggi come Java, non c'è nessun tipo di overhead o virtual tables. Java per esempio riesce a implementare i tipi generici utilizzando delle virtual tables dove il tipo generico viene trattato come oggetto base per poi essere risolto a runtime. Nell'esempio precedente viene creata una sola istanza di template per il tipo *int*. Il sistema di *template* in C++ è turing-completo, nel senso che può eseguire una serie di istruzioni condizionali e ricorsive di complessità arbitraria. Avvalersi della metaprogrammazione attraverso template rende la creazione di librerie di basso livello più facile e consente di crearle efficienti e flessibili.

4.1.2.2 Prestazioni

C++ è considerato un linguaggio molto performante e efficiente, parte delle ragioni sono descritte di seguito.

Strutture dati ottimizzate Le strutture dati che offre C++, e le librerie standard, non hanno nessun tipo di overhead di memoria, quindi per costruzione occupano il minimo indispensabile della memoria che serve per veicolare l'informazione contenuta. Una struttura dati più piccola rappresenta una gestione migliore della cache e della memoria principale.

Compiler ottimizzati C++ è un linguaggio maturo che vanta una grande varietà di compilatori che nel tempo sono stati estremamente ottimizzati per tradurre codice sorgente in codice macchina in maniera efficiente.

Puntatori diretti Come in C, C++ ha la possibilità di referenziare zone di memoria con dei puntatori in modo diretto. Questa possibilità rende il codice potenzialmente insicuro, ma allo stesso tempo dà modo allo sviluppatore di accedere alle risorse velocemente. Recentemente l'avvento degli Smart Pointers (puntatori intelligenti) ha reso l'utilizzo dei puntatori molto più sicuro.

4.2 Librerie e Framework

Per riuscire a simulare casi d'uso complicati come quello descritto in questa tesi sarebbe impensabile sviluppare ogni singolo risolutore di PDE (2.3), convertitore di mesh o sistema di gestione thread. Per ovviare a questo problema ci avvaliamo di alcune librerie Open-Source che altri sviluppatori, matematici, fisici e ingegneri hanno realizzato per risolvere problemi simili. Di seguito alcune librerie che hanno influenzato lo sviluppo della soluzione proposta.

4.2.1 Intel TBB

Intel TBB, acronimo per Threading Building Blocks, è una libreria C++ basata su template (4.1.2.1) che facilita l'implementazione di paradigmi di programmazione parallela [Kuk07]. Un programma che utilizza TBB riesce a creare diversi grafi di task dipendenti fra loro, a organizzarli e sincronizzarli in maniera efficiente e, a task completato, distruggerli. TBB fa uso di una tecnica di bilanciamento di carico parallelo chiamata "work stealing" (letteralmente, furto del lavoro), che bilancia i task che sono in attesa in maniera dinamica verso core scarichi. Il vantaggio di TBB è che fa tutte queste ottimizzazioni in maniera automatica offrendo interfacce di alto livello che rendono semplice l'implementazione. TBB è una libreria che gestisce carichi di lavoro paralleli, ma non garantisce l'esenzione dalle cosiddette *corse critiche* (*race conditions*). TBB offre diversi algoritmi di base:

- `parallel_for`
- `parallel_reduce`
- `parallel_scan`

e avanzati:

- `parallel_while`
- `parallel_do`
- `parallel_pipeline`
- `parallel_sort`

Oltre agli algoritmi sono disponibili diverse strutture e modalità di allocazione dati, operazioni atomiche e funzioni di timing e task scheduling. Vediamo un esempio di come implementare un semplice `parallel_for` sfruttando TBB.

Come prima cosa includiamo le librerie di cui abbiamo bisogno:

```
#include "tbb/blocked_range.h" //Range divisibile ricorsivamente
#include "tbb/parallel_for.h" //Algoritmo
#include "tbb/task_scheduler_init.h" // Scheduler di task
#include <iostream>
#include <vector>
```

Dichiariamo la struttura che incapsula il nostro task:

```
struct task {
    task(size_t n): _n(n)    {}
    void operator()() {
        //Istruzioni arbitrarie che vengono eseguite in parallelo
        for (int i=0;i<1000000;++i) {}
        std::cerr << "[" << _n << "];"
    }
    size_t _n;
};
```

La struttura che esegue i task accetta un vettore di task e ha una funzione che splitta i task e li esegue parallelamente:

```
struct executor
{

    executor(std::vector<task>& t)
    :_tasks(t)
    {}

    executor(executor& e,tbb::split)
    :_tasks(e._tasks)
    {}

    void operator()(const tbb::blocked_range<size_t>& r) const {
        for (size_t i=r.begin();i!=r.end();++i)
            _tasks[i]();
    }

    std::vector<task>& _tasks;
};

int main(int,char**) {
    // Vengono automaticamente inizializzati i threads
    tbb::task_scheduler_init init;

    // Creiamo dei task e li inseriamo in un vettore
    std::vector<mytask> tasks;
    for (int i=0;i<1000;++i)
        tasks.push_back(mytask(i));

    // Creiamo l'executor dei task e invochiamo l'algoritmo TBB
    executor exec(tasks);
    tbb::parallel_for(tbb::blocked_range<size_t>(0,tasks.size()),exec);
    std::cerr << std::endl;
```

```
    return 0;  
}
```

4.2.2 MPI

MPI, acronimo di Message Passing Interface, è un protocollo di comunicazione per calcolatori [For94]. È ormai uno standard di comunicazione usato espressamente per lo scambio di informazioni in calcoli paralleli. MPI è una specifica, non una libreria, di conseguenza esistono diverse librerie che la implementano. L'obiettivo della specifica è quello di fornire un paradigma di programmazione parallela che sia pratico, efficiente e portabile. Lo standard non specifica come eseguire i programmi, piuttosto ogni implementazione fornisce diversi strumenti per compilare ed eseguire programmi MPI. In fase di esecuzione ogni programma implementato con MPI ha a disposizione due funzioni per conoscere il numero di processi che partecipano all'esecuzione parallela e il proprio identificativo di processo. Lo standard inoltre definisce come le informazioni debbano essere comunicate ai vari processi, incapsulando l'informazione in un messaggio che contiene il buffer di dati, il numero di elementi da inviare e il tipo di dato. Il messaggio viene inviato indicando un destinatario, individuato attraverso tag, rank (identificativo processo) e comunicatore (gruppo di processi). Esiste un comunicatore di default a cui appartengono tutti i processi chiamato `MPI_COMM_WORLD`. MPI può operare sia in maniera asincrona (sia bloccanti che non) che in sincrona, a discrezione dell'utente. Un protocollo di comunicazione che permette di scalare su più processori o calcolatori una simulazione CFD è di vitale importanza poiché, come accennato in precedenza, la grande quantità di calcoli richiesti rende i tempi di esecuzione molto lunghi anche per piccole simulazioni con pochi elementi.

4.2.2.1 OpenMPI

OpenMPI è una popolare implementazione del protocollo MPI, sviluppata e mantenuta da un consorzio di accademici, ricercatori e partner. Alcune caratteristiche che rendono OpenMPI un'ottima libreria che implementa MPI:

- Alto livello di thread-safety e concorrenza
- Possibilità di generazione processi in modo dinamico
- Tolleranza in caso di errori di rete o processo
- Compatibile con una grande varietà di sistemi operativi
- Prestazioni elevate su ogni piattaforma supportata
- Portabile e estendibile
- Sviluppo e supporto dalla community attivo
- Licenza open-source



Figura 4.2: Logo OpenMPI

Di seguito un piccolo esempio di come utilizzare la libreria di OpenMPI che stampa a schermo il numero di processori di un calcolatore e il loro rango. Come prima cosa includiamo le librerie necessarie:

```
#include <mpi.h>
#include <stdio.h>
```

All'interno di una funzione `int main(int argc, char** argv)` inizializziamo MPI con:

```
MPI_Init(NULL, NULL);
```

Ricaviamo il numero di processi con:

```
int world_size;  
MPI_Comm_size(MPI_COMM_WORLD, &world_size);
```

Prendiamo il rank del processo:

```
int world_rank;  
MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);
```

Stampiamo e liberiamo le risorse MPI:

```
printf("Processore %d su %d", world_rank, world_size);  
MPI_Finalize();
```

Per far sì che il programma venga eseguito in parallelo, lo compiliamo normalmente e lo eseguiamo così:

```
mpirun -n NUMERO_PROCESSORI NOME_FILE
```

che stamperà su schermo una riga per ogni numero di processori passato all'istruzione `mpirun`.

4.2.3 OpenFoam

OpenFoam, ovvero Open Field Operation And Manipulation [WTJF98], è una libreria C++ che offre una serie di strumenti per sviluppare software di simulazione. OpenFoam è un finalizzatore (chiamato anche Solver) che si basa sulla teoria della meccanica dei continui e che include moduli di fluidodinamica computazionale. OpenFoam offre strumenti per simulare situazioni derivanti da diversi ambiti fisici. Una serie, non esaustiva di strumenti che OpenFoam fornisce:

- Operazioni su tensori e campi

- Discretizzazione di equazioni alle derivate parziali
- Soluzione di sistemi lineari
- Soluzione di equazioni alle derivate ordinarie
- Parallelizzazione automatica di operazioni di alto livello
- Gestione di mesh dinamiche

Questi strumenti sono utilizzati per risolvere i seguenti modelli:

- Modelli reologici
- Modelli termodinamici
- Modelli di turbolenza
- Reazioni chimiche e modelli cinetici
- Tracciamento di particelle lagrangiane
- Modelli di trasferimento di calore per radiazione
- Metodi a singolo e multiplo sistema di riferimento

Tra i tutti i solver che sono inclusi in OpenFoam ricordiamo:

- Fluidodinamica computazionale
- Flussi incomprimibili
- Flussi comprimibili
- Dinamica dei solidi
- Combustione

OpenFoam offre anche diverse utility per la generazione, la conversione e la manipolazione delle mesh, parallelizzazione, pre/post-processing. Un punto di forza che contraddistingue OpenFoam è la possibilità che da all'utente di creare Solvers autonomamente. Purtroppo OpenFoam, però, presenta diversi tipi di problemi che ne rendono l'utilizzo

estremamente difficile. In primis la documentazione è molto scarna, soprattutto quando si tratta di lavorare con delle mesh particolarmente complesse. La curva di apprendimento di OpenFoam proprio per questo motivo è estremamente ripida. La documentazione pressoché assente spesso implica dei costi nascosti quali: ore di lavoro buttate, corsi di aggiornamento o addestramento e consulenze esterne. Quest'ultime sono molto costose proprio per la criticità del settore e impattano l'esperienza open source del prodotto. Alcune scelte implementative effettuate da OpenFoam sono obiettivamente considerate in contrasto con le convenzioni di programmazione, soprattutto in ambito C/C++, un esempio lampante sono le direttive include all'interno del corpo delle funzioni. Infine un ulteriore motivo per cui OpenFoam è stata scartata è l'impossibilità di formulare un problema in cui si accoppia fluido e particelle integrando il campo della temperatura.

Vediamo un piccolo esempio di programma che interpreta una mesh utilizzando OpenFoam:

```
#include "fvCFD.H"

int main(int argc, char *argv[])
{
    #include "setRootCase.H"

    //Qui creiamo il time system e l'istanza della mesh.
    //notare le direttive di include nel corpo della funzione.
    #include "createTime.H"
    #include "createMesh.H"

    // Iteriamo le celle della mesh e stampiamone a schermo 1 su 20
    for (label cellI = 0; cellI < mesh.C().size(); cellI++)
        if (cellI == 0)
            Info
            << "Cella: "
            << cellI
```

```
<< " centrata in: "  
<< mesh.C()[cellI]  
<< endl;  
Info << endl;  
  
// Ogni cella è costituita da facce interne o di contorno  
for (label faceI = 0; faceI < mesh.owner().size(); faceI++)  
    if (faceI == 0)  
        Info  
        << "Faccia interna: "  
        << faceI  
        << " centrata in "  
        << mesh.Cf()[faceI]  
        << " cella padre " << mesh.owner()[faceI]  
        << " cella vicina " << mesh.neighbour()[faceI] << endl;  
Info << endl;  
  
// Le condizioni al contorno sono accedute  
// attraverso boundaryMesh().  
// OpenFOAM fornisce dei for custom per ridurre il numero  
// di codice da scrivere.  
forAll(mesh.boundaryMesh(), patchI)  
    Info  
    << "Patch "  
    << patchI  
    << ": "  
    << mesh.boundary()[patchI].name() << " con "  
    << mesh.boundary()[patchI].Cf().size() << " facce. "  
    << mesh.boundary()[patchI].start() << endl;  
Info << endl;
```

```

// Le facce adiacenti a dei contorni sono accedute così.
label patchFaceI(0);
forAll(mesh.boundaryMesh(), patchI)
    Info << "Patch "
        << patchI
        << " ha una faccia "
        << patchFaceI
        << " adiacente alla cella "
        << mesh.boundary()[patchI].patch().faceCells()[patchFaceI]
        << ". Normale " << mesh.boundary()[patchI].Sf()[patchFaceI]
        << " superficie "
        << mag(mesh.boundary()[patchI].Sf()[patchFaceI])
        << endl;
Info << endl;

// Si possono iterare anche i singoli
// punti o vertici della mesh
const faceList& fcs = mesh.faces();
const List<point>& pts = mesh.points();
const List<point>& cents = mesh.faceCentres();

forAll(fcs,faceI)
    if (faceI%80==0)
    {
        if (faceI<mesh.Cf().size())
            Info << "Internal face ";
        else
        {
            forAll(mesh.boundary(),patchI)
                if ((mesh.boundary()[patchI].start()<= faceI) &&
                    (

```

```
        faceI <- mesh
        .boundary()[patchI]
        .start()+mesh
        .boundary()[patchI]
        .Cf()
        .size())
    )
  {
    Info << "Faccia su Patch "
    << patchI
    << ", faceI ";
    break;
  }
}

Info << faceI << " centrata in " << cents[faceI]
    << " ha " << fcs[faceI].size() << " vertici:";
forAll(fcs[faceI],vertexI)
  Info << " " << pts[fcs[faceI][vertexI]];
Info << endl;
}
Info << endl;

return 0;
}
```



Figura 4.3: Logo OpenFoam

4.2.4 FEniCS Project

FEniCS è una piattaforma open-source il cui scopo è il calcolo di equazioni alle derivate parziali [LORW12]. FEniCS utilizza un interfaccia di alto livello scritta in Python che facilita l'implementazione a chi non ha esperienze pregresse di programmazione. FEniCS fa largo uso di librerie C++ che offrono a chi usa questa piattaforma elevate prestazioni. FEniCS è ottimizzata per essere usata su anche su cluster di calcolatori molto performanti. Nonostante sia una piattaforma molto popolare per la risoluzione PDE (2.3) attraverso metodo FEM, l'idea di implementare un simulatore di flussi particellari è stata scartata, principalmente per problemi di compatibilità fra diversi sistemi, ma non solo. Per quanto sia possibile utilizzare Anaconda 4.3.1.1 per risolvere le dipendenze dei pacchetti usati da Fenics, questo spesso non garantisce il corretto funzionamento su piattaforme uguali. Inoltre nonostante sia una libreria in circolazione dal 2005 viene spesso aggiornata deprecando intere librerie di computazione matematica tra una versione e l'altra. Oltre a ciò le interfacce fra C++ e Python non sono ancora propriamente separate, il namespacing non è intuitivo, la parallelizzazione non sfrutta a pieno le risorse computazionali messe a disposizione e la documentazione è ancora poco dettagliata.

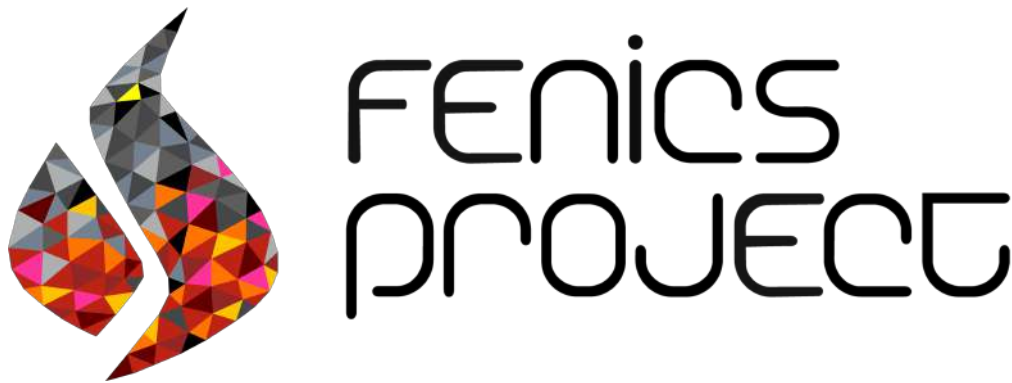


Figura 4.4: Logo Fenics

4.2.4.1 LEOPart

LEOPart (Langrangian-Eulerian on Particles) integra alcune funzionalità riguardanti i flussi di particelle in FEniCS [MRS19]. L'obiettivo della libreria è quello di occuparsi di avvezione e proiezione di particelle in una mesh in modo accurato e conservativo. La libreria è stata scartata poiché ha subito diverse modifiche durante lo sviluppo di questo progetto, inoltre integrare la simulazione della temperatura del fluido risultava particolarmente complicato.

4.2.5 deal.II

Deal.II, o Differential Equations Analysis Library versione 2 (II), è una libreria che ha come obiettivo la risoluzione di equazioni differenziali alle derivate parziali attraverso l'utilizzo del metodo degli elementi finiti [ABB⁺20]. È una libreria moderna e open-souce che fornisce un'interfaccia semplice capace di sfruttare complessi algoritmi e strutture dati per risolvere problemi numerici, il tutto in maniera molto efficiente e performante. Questa libreria, alla base di molti strumenti di simulazione moderni, rende il processo di sviluppo rapido e sicuro, offrendo una serie di funzionalità ... [aggiungi funzionalità e funzionamento]

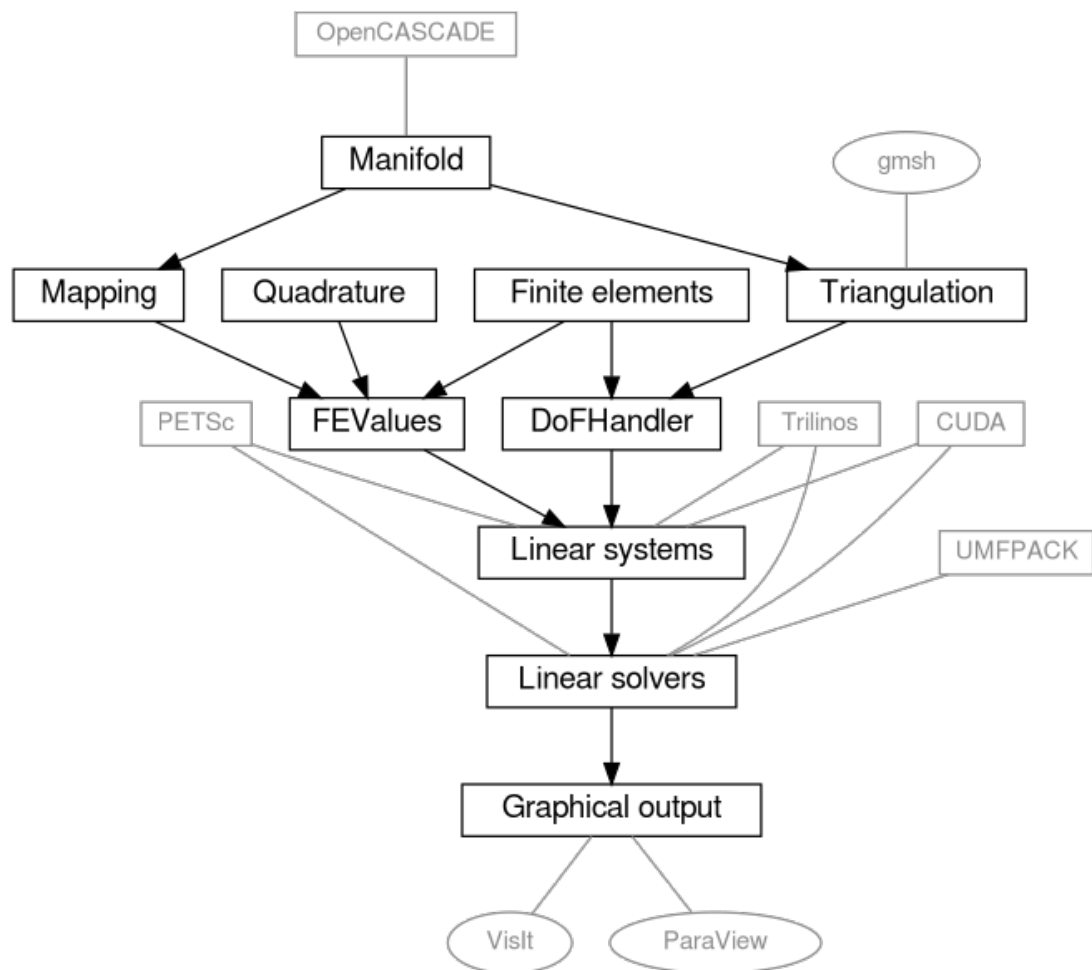


Figura 4.5: Architettura deal.II

4.2.6 Aspect

Aspect, acronimo per Advanced Solver for Problems in Earth's ConvecTion [BDG⁺20], è un programma open-source sviluppato in C++ che supporta la ricerca simulando diversi tipi di scenari tra cui:

- Convezione del mantello terrestre
- Convezione di metalli vicini al nucleo terrestre
- Deformazione litosferica
- Flussi bifasici

Aspect ha avuto un impatto importante nello sviluppo di questo progetto, essendo un programma che simula problemi di convezione termica ha contribuito in alcune scelte tecniche che discuteremo più avanti. Aspect fa largo uso di deal.II per creare e risolvere problemi descritti da equazioni differenziali alle derivate parziali attraverso l'uso del metodo degli elementi finiti.

4.3 Strumenti di sviluppo

Per ovviare alle difficoltà che si presentano in fase di sviluppo di progetti con molteplici dipendenze e che si avvalgono dell'utilizzo di altrettante librerie di terze parti, è altamente consigliato progettare un ambiente di sviluppo che faciliti la produzione di applicazioni software. Ogni linguaggio di sviluppo ha un particolare processo di conversione di codice ad eseguibile (4.1.2), inoltre anche la sintassi varia molto da linguaggio a linguaggio. Gli strumenti di sviluppo che andremo ad analizzare nelle prossime sezioni risultano quindi di notevole ausilio all'utente in ogni fase dello sviluppo, dalla stesura del codice al confezionamento dell'applicativo.

4.3.1 Gestione Librerie e pacchetti

4.3.1.1 Anaconda

Anaconda è una particolare distribuzione del linguaggio Python (4.1.1) specializzata nella gestione di pacchetti, dipendenze e rilasci software. Le versioni dei pacchetti in

Anaconda sono gestite attraverso *conda*, un gestore pacchetti, separato dalla suite Anaconda, molto popolare nella comunità Python. Conda svolge anche il compito di gestore di ambienti virtuali, ovvero può creare degli ambienti separati al cui interno vengono rese disponibili determinati pacchetti definiti dall'utente agli applicativi. Questo garantisce che ogni eseguibile all'interno dell'ambiente virtuale abbia correttamente impostate le varie dipendenze di cui necessita per operare. Di seguito un comando di esempio per l'installazione di Fenics (4.2.4) e l'attivazione di un ambiente che ne comprende tutte le dipendenze.

```
conda create -n fenicsproject -c conda-forge fenics
source activate fenicsproject
```

Anaconda è gratuito e open-source, sviluppato però da Anaconda,Inc. che offre anche particolari versioni enterprise della distribuzione.

4.3.2 IDE

Un IDE, acronimo per *integrated developmnet environment*, rappresenta un ambiente di sviluppo al cui interno sono integrati una serie di strumenti che aiutano il programmatore a: scrivere, testare, debuggare, compilare codice di uno o più linguaggi di programmazione. Nella fase di scrittura del codice l'IDE fornisce assistenza segnalando errori di sintassi, convenzioni di nomenclatura (specifiche per il particolare linguaggio in uso) e suggerendo simboli e variabili da utilizzare in base al contesto. Molti IDE offrono anche delle suite di test che facilitano la scrittura e l'esecuzione di test automatici. Spesso gli IDE possiedono uno o più ambienti di debugging, che aiutano lo sviluppatore a eseguire le istruzioni del programma in maniera controllata e a comando, in modo tale da garantire una visibilità più ampia su ciò che accade in fase di esecuzione del programma. Inoltre gli IDE si avvalgono di uno o più compilatori per confezionare gli eseguibili con facilità.

4.3.2.1 Clion

Clion è un IDE cross-platform specifico per linguaggi quali C e C++ sviluppato da JetBrains. Viene considerato uno *smart editor* in quanto: suggerisce in tempo reale refattorizzazioni e generazioni del codice automatiche, analizza automaticamente il codice per eventuali regole non rispettate (sia di convenzioni che di buone pratiche di sviluppo), gestisce progetti di sviluppo, ha un debugger integrato e un sistema di versionamento del codice integrato.

Clion può utilizzare diversi compilatori sia per C che per C++, inoltre supporta CMake e Make (4.3.3.2 e 4.3.3.1). Una funzionalità che si è rilevata particolarmente utile durante lo sviluppo di questo progetto è chiamata *full remote mode*, che permette di sfruttare Clion come IDE utilizzando come ambiente di sviluppo una macchina separata rispetto a quella su cui risiede l'IDE. Questa macchina può essere anche un container (più informazioni sui container di seguito 4.3.4)

4.3.3 Compilatori e altri strumenti

Come anticipato in 4.1.2 ecco una serie di validi strumenti che facilitano lo sviluppo di applicazioni C++ e che sono stati usati assiduamente durante lo sviluppo di questa tesi.

4.3.3.1 Make

Make è uno strumento sviluppato originariamente su sistemi operativi UNIX da Stuart Feldman, ora in mano alla GNU Operating System (parte della Free Software Foundation), che automatizza il processo di creazione di file dipendenti da altri file. Un'automatizzazione di questo tipo è molto usata nella compilazione del codice sorgente. Make si avvale di *makefile* principalmente per due scopi:

- dichiarare l'ordine e il grado di dipendenze di file per un particolare output
- invocare script necessari alla compilazione da passare alla shell che li esegue

In progetti che si avvalgono di molte librerie di terze parti, o che hanno una struttura di file particolarmente distribuita e complessa, sarebbe quasi impossibile collegare i vari

file fra di loro rispettando le corrette dipendenze senza utilizzare uno strumento come Make.

Nonostante Make sia stato sviluppato nel 1977 le sue derivazioni sono ancora utilizzate oggi.

4.3.3.2 CMake

CMake rappresenta una serie di strumenti cross-platform sviluppati per gestire le fasi di compilazione, test e distribuzione del software. CMake viene istruito attraverso istruzioni contenute in uno o più file di configurazioni (il principale chiamato CMakeLists.txt) indipendenti dal compilatore scelto. CMake è in grado di generare automaticamente, a partire dai file di configurazione, i makefile necessari alla compilazione del programma. Oltre alla generazione dei makefile un altro compito principale di CMake è quello di verificare che tutte le librerie necessarie alla compilazione del programma siano presenti ed utilizzabili. Di seguito un semplice esempio di file di configurazione CMake:

```
PROJECT(flows)
ADD_DEFINITIONS(-pipe -O2 -mtune=native)
ADD_EXECUTABLE(bin/flows src/main.cpp)
```

Dato un CMakeLists.txt correttamente configurato, per generare il makefile e compilare il progetto basta eseguire:

```
mkdir build
cd build
cmake ..
make
```



Figura 4.6: Logo CMake: CMake è distribuito in modalità open-source da Kitware.

4.3.3.3 GCC

GCC, o *GNU Compiler Collection*, è un compilatore multipiattaforma che fa parte del progetto GNU. Originariamente fu creato da Richard Matthew Stallman, fondatore della Free Software Foundation, nel 1987. Inizialmente nacque come compilatore C, ma nel tempo è stato aggiunto il supporto a altri linguaggi come C++, Objective C, Ada, Go e altri. Il compilatore è in grado di generare linguaggi macchina per diverse architetture tra cui x86, x86-64 e ARM (recentemente diventato proprietà di Nvidia). Essendo un compilatore ha come scopo primario la traduzione da codice sorgente a codice macchina eseguibile. Il funzionamento del compilatore è già stato descritto nella sezione 4.1.2, ma un esempio di compilazione è il seguente:

```
gcc main.c
```

Il risultato è un file eseguibile che viene chiamato *a.out* (è possibile specificare un nome di output attraverso l'opzione *-o* di gcc).

4.3.3.4 GDB

GDB, acronimo per *GNU Debugger*, è un programma di debugging open-source sviluppato ancora una volta dal progetto GNU. Può essere eseguito su molte piattaforme ed è il debugger predefinito del sistema operativo GNU. Il debugger può analizzare in fase di esecuzione (*runtime*) le istruzioni di codice in maniera controllata e fornendo diverse informazioni all'utente tra cui:

- valore delle variabili o degli oggetti referenziati
- stack trace delle chiamate effettuate
- utilizzo memoria

Per utilizzare il GDB è necessario compilare il programma fornendo il flag corretto di debug (nel caso di gcc *-g*), questo comporta la creazione di un eseguibile che contiene informazioni di debugging aggiuntive necessarie a GDB per funzionare correttamente. Di seguito un Comando per eseguire gdb su un eseguibile compilato:

`gdb a.out`

Il debugging è particolarmente utile se si utilizza in congiunzione ai *breakpoint* che, se posti in corrispondenza di una particolare istruzione, bloccano l'esecuzione prima dell'istruzione selezionata.

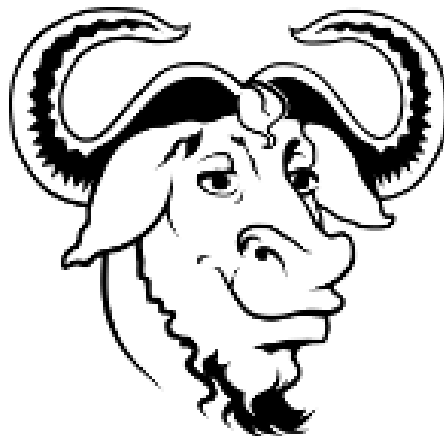


Figura 4.7: Logo GNU Operating System

4.3.4 Docker

Docker è un progetto open-source sviluppato da Docker, Inc. disponibile dal 2013. Docker automatizza il rilascio e l'esecuzione di applicazioni all'interno di contenitori software. Fornisce quindi un'astrazione aggiuntiva grazie alla virtualizzazione a livello di sistema operativo. Docker infatti utilizza le funzionalità di isolamento del kernel linux per consentire ai container di esistere in maniera indipendente l'uno dall'altro. Docker risulta molto utile anche per risparmiare spazio fisico sia sulla memoria principale in fase di esecuzione dei container, sia sulla memoria secondaria in termini di dimensione di immagini virtuali. Utilizzando i namespace del kernel linux si raggiunge l'isolamento tra container mentre con *cgroup* si ottiene l'isolamento in termini di risorse del calcolatore. Nella figura 4.8 è possibile visualizzare le differenze tra le comuni macchine virtuali e i container di Docker.

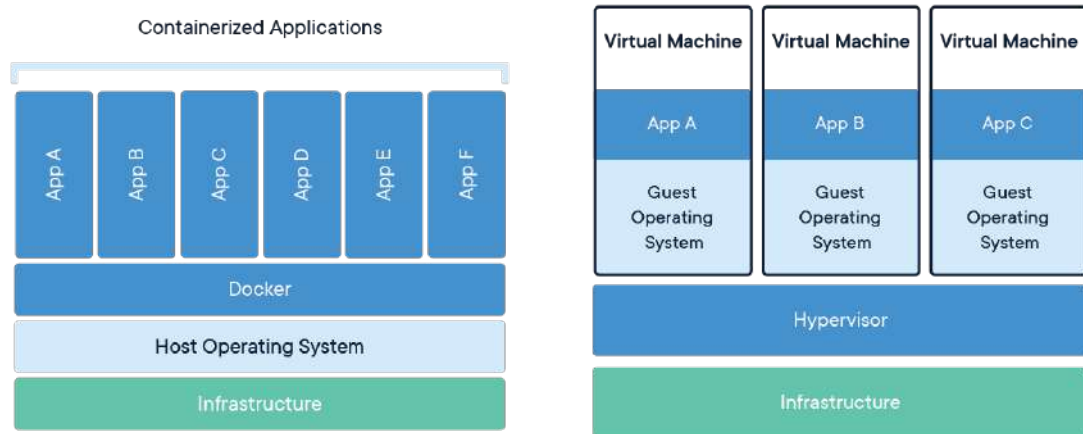


Figura 4.8: Differenze fra macchine virtuali e container Docker

Docker svolge il proprio compito grazie a più componenti che cooperano fra loro (4.9). Il *Docker Engine* è un applicazione client-server composto da:

- Servizio di tipo *daemon* (servizio a lunga durata) composto da un server.
- API REST con la quale si può comunicare per gestire il daemon citato sopra.
- Interfaccia a linea di comando (invocabile tramite il comando *docker*)

Docker inoltre offre anche un servizio di salvataggio e gestione per le immagini dei container, *Docker Hub*, dando la possibilità agli utenti di versionare i vari rilasci delle immagini che creano.

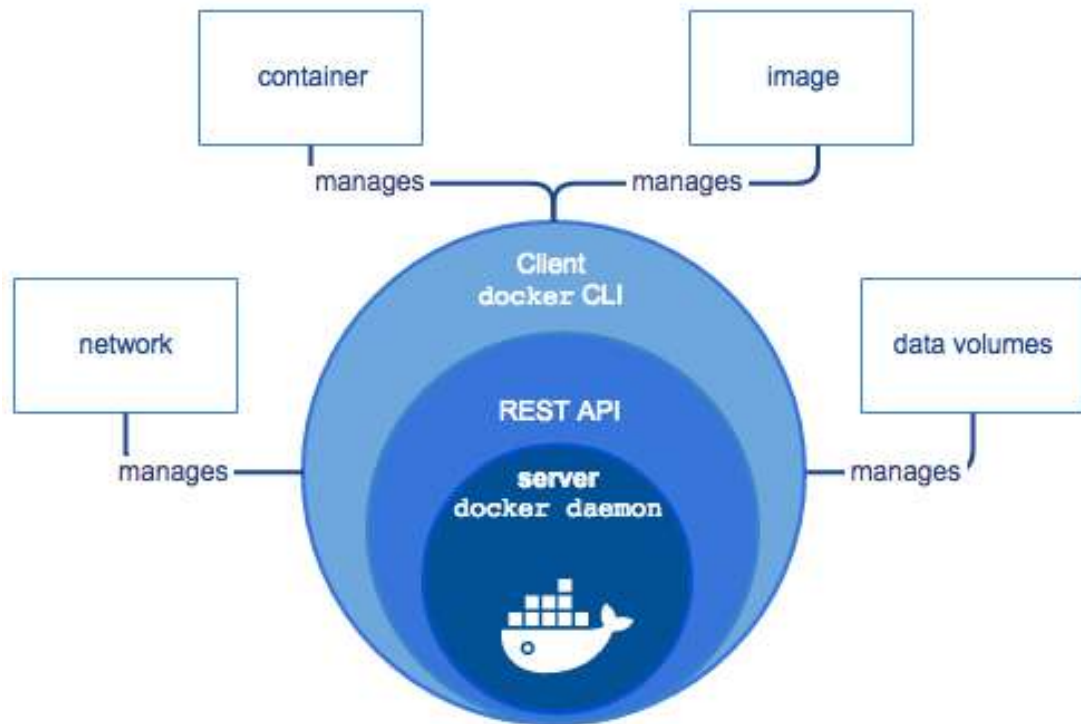


Figura 4.9: Differenze fra macchine virtuali e container Docker

In questo progetto Docker è stato usato principalmente in congiunzione con Clion 4.3.2.1 per utilizzare immagini ufficiali di Deal.II (4.2.5) e risolvere le dipendenze tra i vari componenti della libreria. Sono infatti presenti varie immagini per container di Deal.II al cui interno sono già soddisfatti i vincoli imposti dalle librerie usate (OpenMPI, Tbb, ecc). Docker rende quindi pressoché impossibili problemi di compatibilità di programmi tra diverse piattaforme e sistemi.



Figura 4.10: Logo Docker

4.4 Strumenti di visualizzazione e meshing

Uno degli aspetti fondamentali quando si realizzano delle simulazioni CFD è la corretta costruzione e interpretazione della mesh (3.2.1) su cui operare. Sarebbe impensabile costruire manualmente le mesh necessarie per simulare complessi condotti o ugelli, per questo motivo ci affidiamo a strumenti di meshing creati appositamente per semplificare questo processo.

Le simulazioni di questo tipo, inoltre, generano una grande quantità di dati che possono risultare complicati da interpretare. Nel caso di una simulazione di particelle infatti per ogni punto vengono salvate informazioni riguardo la posizione (un dato per ogni asse geometrico presente), la velocità (sempre per ogni asse geometrico) e temperatura per esempio. Questi dati vengono salvati per ogni particella e per ogni intervallo di tempo discretizzato (time-step), si intuisce quindi che la mole di dati con cui si ha a che fare in questi casi è importante. Spesso inoltre questi dati vengono rappresentati secondo diversi formati, è quindi importante utilizzare un software di visualizzazione capace di interpretare questi dati e mostrarli in maniera intuitiva.

4.4.1 Paraview

Paraview è un'applicazione open-source, multi-piattaforma di analisi e visualizzazione dati. Permette di creare visualizzazioni 2D o 3D in modo interattivo o programmatico, grazie alle sue funzionalità di processamento batch. Paraview può anche essere dispiegato in cluster di calcolatori per analizzare quantità di dati nell'ordine dei Peta-Byte. Paraview ormai è considerato uno standard nel mondo scientifico ed è usato in molti

contesti differenti, sia a livello universitario che industriale. All'interno di questo progetto, Paraview è stato usato massivamente per tracciare il movimento delle particelle e soprattutto per verificare le condizioni di rimbalzo.

4.4.1.1 Paraview e OSPRay

Di recente Paraview ha aggiunto il supporto alla libreria di ray tracing sviluppata da Intel OSPRay [WJA⁺17], che permette di realizzare dei render grafici sfruttando le tecniche di geometria ottica basate sul calcolo dei percorsi fatti dai fotoni stessi e dell'interazione fra essi e materiali realistici rispetto al punto di vista di una telecamera.

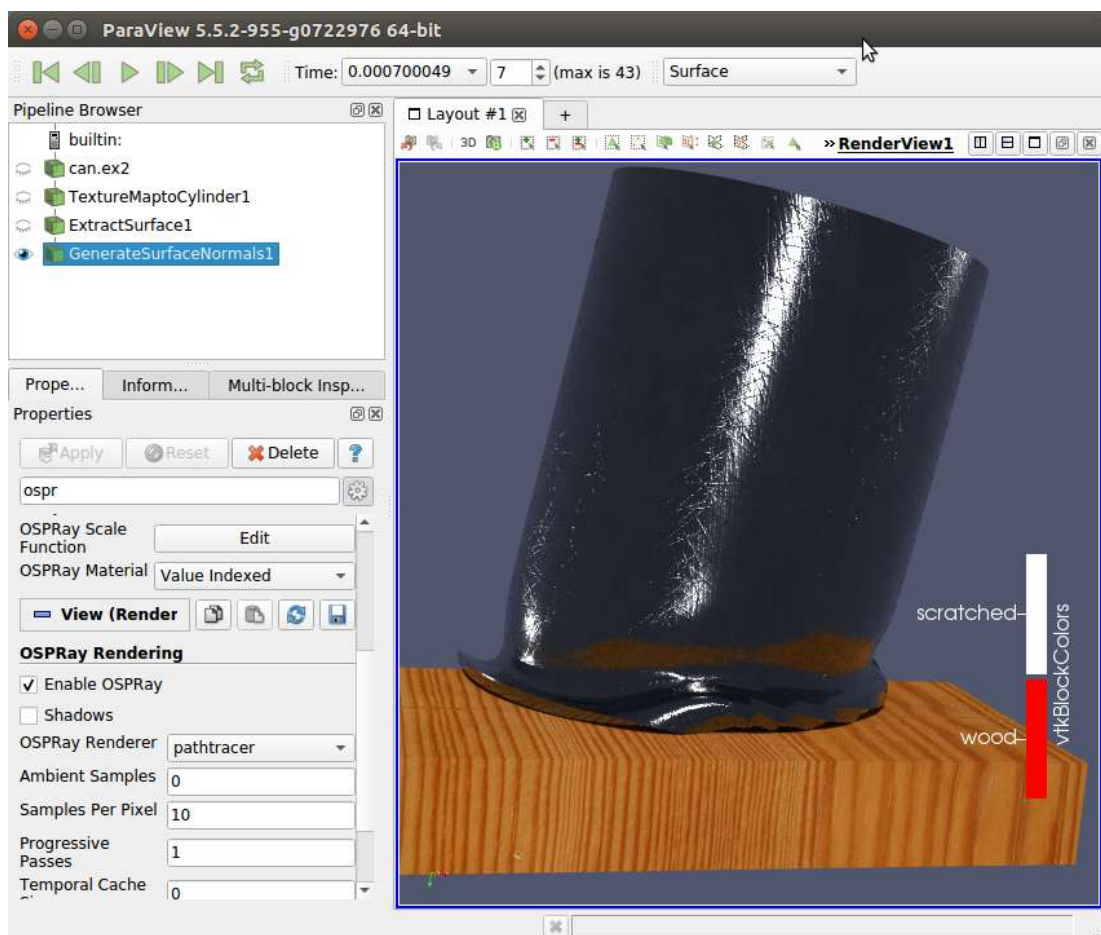


Figura 4.11: Deformazione renderizzata con OSPRay in Paraview



Figura 4.12: Logo Paraview

4.4.2 SALOME

SALOME è una suite di strumenti open-source che fornisce moduli di pre e post processing per simulazioni, meshing e calcoli numerici su modelli. SALOME offre una serie di componenti che facilitano la creazione, modifica, importazione e esportazione di modelli CAD, generando per questi ultimi mesh in svariati formati comprensibili da altrettanti strumenti di CFD (compreso dealii 4.2.5). Un punto di forza di SALOME è la disponibilità di un interfaccia intuitiva che rende il processo di meshing molto più semplice e veloce. Durante lo svolgimento di questa tesi SALOME è stato utilizzato per generare la mesh finale dell'ugello responsabile per il deposito di materiale.

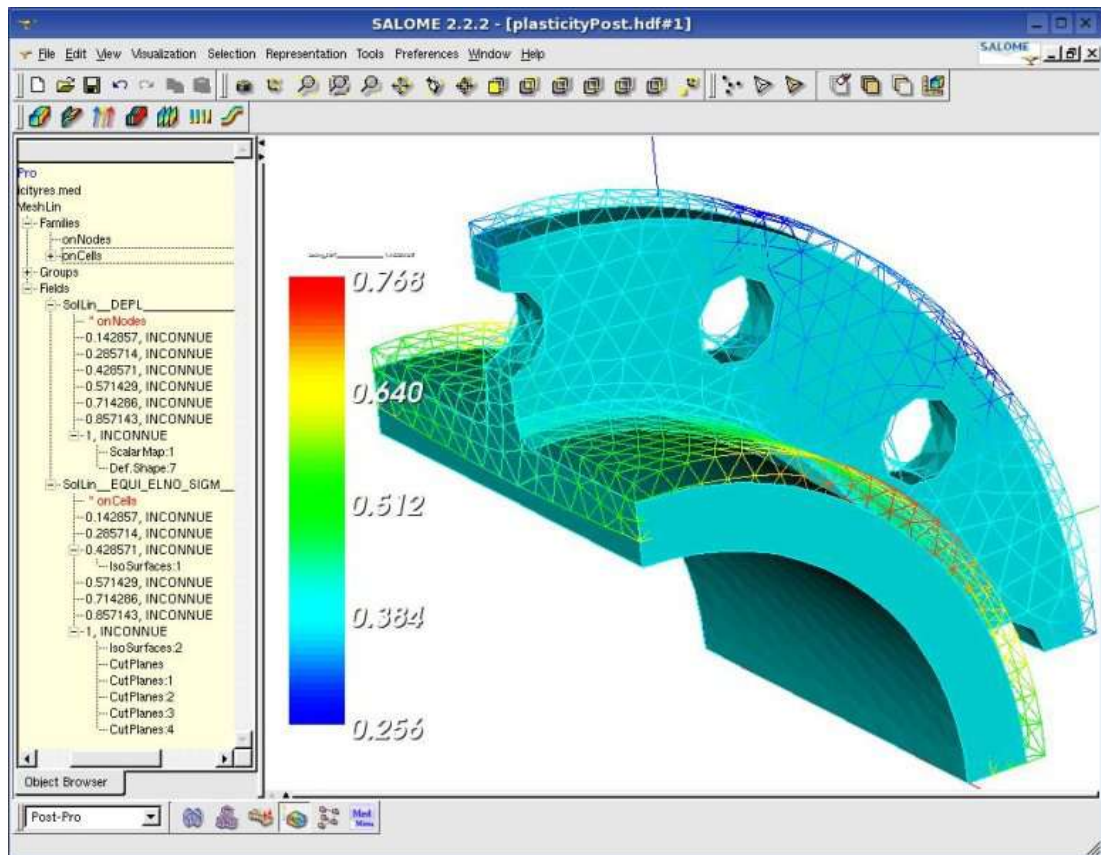


Figura 4.13: Interfaccia SALOME

Capitolo 5

Solutore Computazionale

Realizzare simulazioni di processi LMD senza avvalersi di software CFD sarebbe pressoché impossibile. Al momento il settore dei software per CFD è ricco di strumenti e applicazioni, che attraverso processi semplificati e interfacce intuitive, rendono molto più veloce e semplice tutto ciò che concerne il ciclo di progettazione di soluzioni che necessitano analisi e test di flussi, deformazioni e molto altro. Il problema principale che spesso è causa dell'inutilizzo di questi programmi è il costo di utilizzo in termini economici, non è raro infatti imbattersi in licenze che superano anche i 50000 Euro/Anno per postazione. Come abbiamo visto nel capitolo precedente esistono anche soluzioni open-source, molto popolari in ambito accademico, che però richiedono conoscenze informatiche, oltre che matematiche e fisiche, avanzate per essere utilizzate correttamente. Avere a disposizione il codice sorgente di uno strumento capace di risolvere PDE (2.3) con FEM (2.3) o FVM è un grande vantaggio, dà la possibilità di comprendere meglio il funzionamento di simulazioni complesse e modificarlo a proprio piacimento. In questo capitolo analizzeremo ciò che è stato prodotto da un punto di vista software, per soddisfare gli obiettivi prefissati. In particolare discuteremo le scelte architetturali e i pattern software utilizzati, motiveremo le strutture dati e le librerie impiegate.

Lo sviluppo del simulatore ha subito diversi cambiamenti radicali durante lo svolgimento di questo percorso. Per ovviare alle difficoltà implementative che questi cambiamenti ponevano si è subito cercato di compartimentalizzare ogni macro modulo di funzionalità per rendere più facile, veloce e collaborativo lo sviluppo. Ogni modulo è

orchestrato da un punto di ingresso (o **main**) che inizializza e coordina gli stessi al fine di produrre in output dei risultati. Solitamente per simulare un evento fisico che si svolge in un intervallo di tempo continuo lo si discretizza introducendo il concetto di **time-step**. Il **time-step** è ciò che determina la risoluzione temporale della simulazione, ovvero il lasso di tempo che passa tra un intervallo e il successivo. Per esempio in un lasso di tempo di 10 secondi con un **dt** (delta temporale) di 0.1s secondi otteniamo:

$$\frac{10s}{0.1s} = 100 \text{ time-step}$$

I **time-step** vengono utilizzati all'interno della simulazione per determinare quante volte viene eseguito il passo risolutivo di ogni modulo.

Di seguito è mostrato un diagramma dei moduli e la loro interazione:

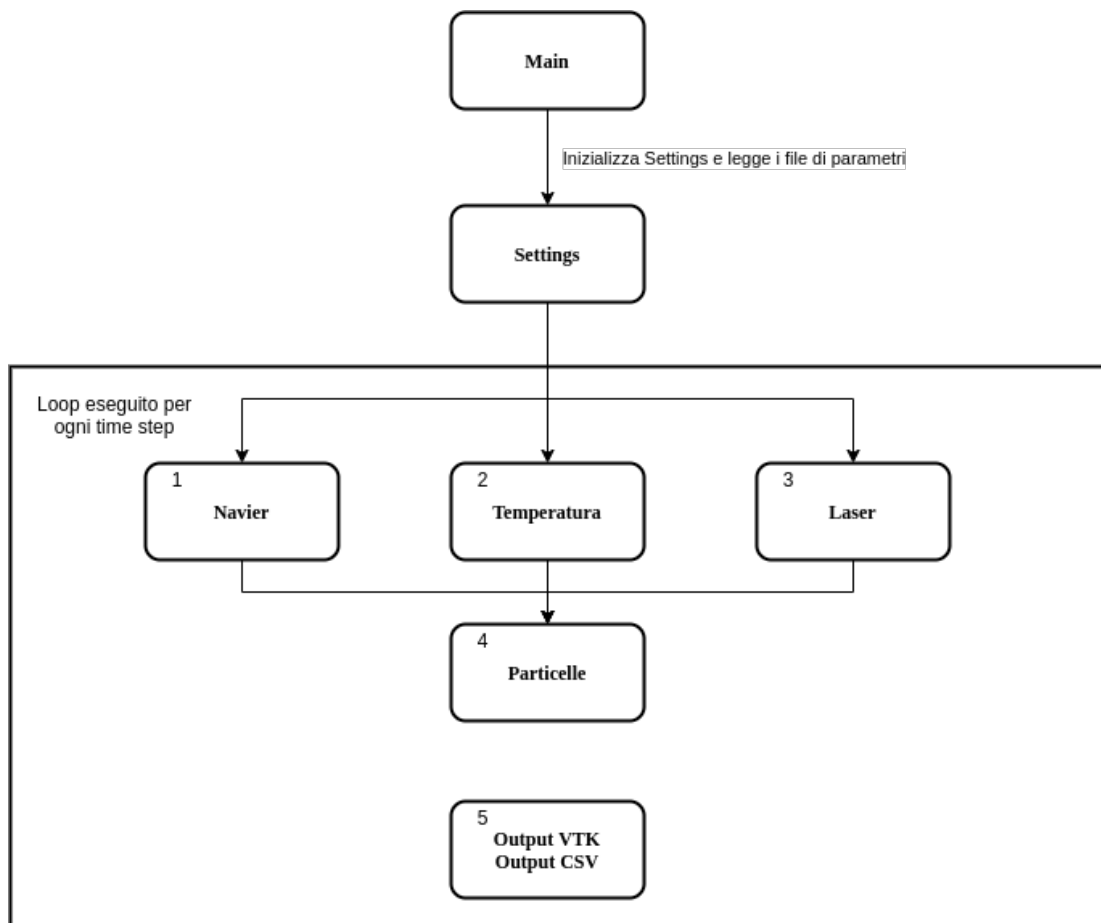


Figura 5.1: Interazione fra moduli.

Personalmente all'interno del progetto ho partecipato più attivamente allo sviluppo del modulo Particellare (5.6), ma di seguito è riportata una breve descrizione anche degli altri moduli. I moduli sono ordinati su base temporale, quindi dal primo utilizzato all'ultimo, viene inoltre fornito un diagramma di flusso per ognuno.

5.1 Gestione Input e impostazioni

Per rendere il software adatto a diverse simulazioni in maniera intuitiva e veloce, è stato creato un modulo responsabile della gestione dei parametri iniziali di simulazione. Il modulo si avvale dell'interfaccia esposta dalla libreria `deal.II` (4.2.5) `deal.II/base/parameter_handler.h`, che facilita la gestione di parametri e il relativo parsing da file (`parameter_file.prm`).

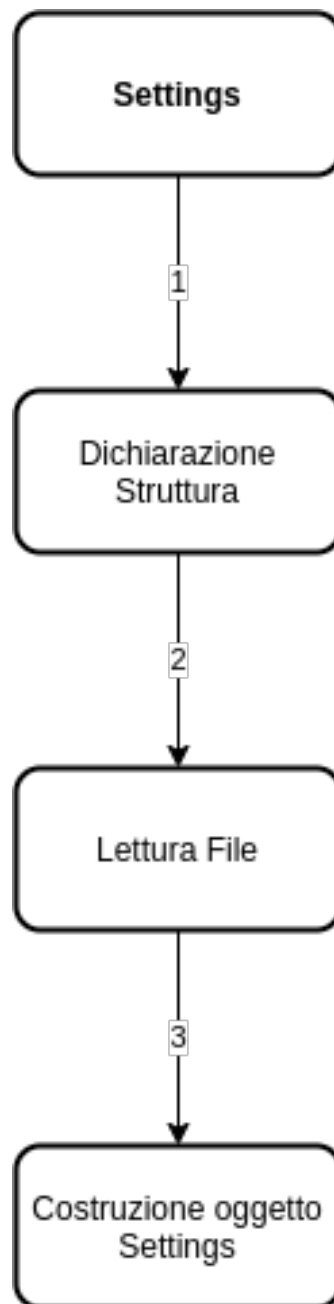


Figura 5.2: Flusso di gestione input.

5.1.1 Struttura file di input

Il file di parametri ha una struttura composta di diverse **section**, caratterizzate da un nome, e molteplici variabili che possono rappresentare sia numeri che stringhe. Le section possono anche essere annidate una dentro l'altra per avere a disposizione un ulteriore livello di organizzazione dei parametri.

```
section Simulation parameters
    set simulation_time      = 0.2
    set time_step            = 0.1
    set theta                = 1.0
    set mesh_file            = ../mesh/mesh.dealii
    set refinement_mesh_degree = 1
end
```

Analizzando la sottosezione del file di parametri si può notare come non vengono utilizzati nessun tipo di identificatore per distinguere i vari tipi di dato (numeri interi, decimali o stringhe), bensì tratta tutto ciò che viene posto dopo il segno di uguaglianza come semplici caratteri. Sarà infatti compito di 5.1 distinguere i vari tipi in base alle istruzioni di dichiarazione in fase di parsing, istruzioni che vedremo nel dettaglio in 5.1.2. Il file può contenere un numero arbitrario di sottosezioni, è buona pratica però organizzare le sottosezioni in maniera analoga a quella del resto del progetto; ovvero compartimentalizzando le impostazioni rispettivamente alle proprie aree di interesse. Di seguito è rappresentata la struttura finale del file utilizzato per raggiungere i risultati di 6.

```
section Simulation parameters
    set simulation_time      = 0.1
    set time_step            = 0.0002
    set theta                = 1.0
    set mesh_file            = ../mesh/mesh.unv
    set refinement_mesh_degree = 0
end
```



```
section FEM parameters
    set fe_degree          = 1
end

section Physical parameters

    #Boundary
    set inlet_manifold = 1
    set outlet_manifold = 2
    set walls_manifold = 4
    set laser_manifold = 3
    set substrate_manifold = 5
    set time_dependent_boundary = false

    # Geometry
    set l_channel          = 0.0227
    set h_channel          = 0.001236
    set substrate_location = -0.0077

    # Navier-Stokes
    set inlet_velocity_x   = 0.0
    set inlet_velocity_y   = 0.0
    set inlet_velocity_z   = -1.5
    set inlet_type         = constant
    set reynolds_number    = 100
    set strouhal_number    = 1.0

    #Temperature
    set prandtl_number     = 0.1
    set minimum_temperature = 50.0
```

```
set maximum_temperature      = 1500.0
set plateau_temperature      = 0.000001

end

section Particle data
    set particle_density      = 8000
    set particle_diameter     = 0.00005
    set inlet_particle_mass   = 0.0012
    set inlet_duration        = 0.01
    set drag_coefficient      = 0.47
    set particle_specific_heat = 470.0
    set particle_absorptivity_rate = 0.4
    set particle_convection_coefficient = 20.0
    set fluid_density         = 1.2506
    set gravity                = 0.0
    set initial_particle_velocity_x = 0.0
    set initial_particle_velocity_y = 0.0
    set initial_particle_velocity_z = -1.5
    set initial_particle_temperature = 100.0
    set block_particles_on_boundary = true

end

section Laser data
    set laser_power           = 1000.0
    set laser_radius           = 0.003

end

section Solver parameters
```

```
# Linear and non-linear solvers
set algorithm = linear

# Newton settings
set newton_tolerance = 1e-10
set max_newton_iterations = 50

# Projection method settings
set reinit_vel_preconditioner = 10
set max_iterations = 1000
set eps = 1e-8
set Krylov_size = 30
set off_diagonals = 70
set diag_strength = 0.1

# Iterated Projection method settings
set iPC_pressure_tollerance = 1e-3
set iPC_max_pressure_iterations = 100
end

section Output parameters
  set output = 1
  set output_directory = output/
  set output_filename = flow
  set write_to_file = false
end
```

L'esternalizzazione dei parametri della simulazione permette di eseguire il programma con diverse impostazioni di partenza senza dover ricompilare la soluzione ogni volta. Vediamo ora come vengono passate le informazioni dal file di parametri al programma stesso.

5.1.2 Implementazione

La libreria di gestione parametri di dealii espone la classe **ParameterHandler** e le relative funzioni per gestire la struttura dei file di parametri (Si consideri **prm** istanza della classe **ParameterHandler**):

- `enter_section(section_name)`

Imposta la sottosezione su cui opera l'oggetto **prm**, ogni successiva ricerca di variabili (o **entry**) verrà effettuata in questa sottosezione. La sottosezione viene indicata attraverso il nome della stessa, passato come stringa alla funzione.

```
prm.enter_section("Simulation parameters");
```

- `leave_section()`

Una volta "entrati" in una sottosezione si utilizza questa istruzione al fine di reimpostare **prm** al livello di sezione precedente.

```
prm.leave_section();
```

- `declare_entry(name, default_value, pattern, documentation_name)`

Prima di leggere il file di parametri dobbiamo istruire **prm** sulla struttura da interpretare.

```
prm.declare_entry("simulation_time",  
                  "10.0",  
                  Patterns::Double(0),  
                  "simulation_time");
```

Analizziamo parametro per parametro l'istruzione. Il primo parametro è usato per definire il nome del parametro che **prm** si aspetta di trovare all'interno del file di parametri. A seguire troviamo un valore di default che verrà utilizzato

in caso non si trovi il parametro all'interno del file di parametri, notare che pur trattandosi di un valore di tipo `double` passiamo comunque una stringa. Il terzo parametro rappresenta un pattern testuale utilizzato per convalidare il valore che viene letto da `prm`, in caso suddetto valore non rispetti il pattern selezionato verrà utilizzato il valore di default. Esistono diversi tipi di pattern tra cui `Double`, `Anything (string)` e `Integer`. Infine l'ultimo parametro serve a definire il nome del parametro in un eventuale generazione documentale attraverso l'istruzione `print_parameters()`.

- `get_double(entry_name)`

Come si evince dal nome della funzione attraverso quest'istruzione si può ricavare il valore del parametro trasformato direttamente in tipo `double`.

- `get(entry_name)`

Identicamente alla funzione precedente, con la differenza che il tipo del valore ritornato è una stringa.

- `get_integer(entry_name)`

Identicamente alla funzione precedente, con la differenza che il tipo del valore ritornato è un intero.

Mettendo insieme quanto analizzato finora, ecco un esempio di come dichiarare e leggere un segmento di `parameter_file.prm`

```
ParameterHandler prm;
```

```
prm.enter_section("Simulation parameters");
prm.declare_entry("simulation_time", "10.0", Double(0), "sim_time");
prm.declare_entry("time_step", "0.005", Double(0), "time_step");
prm.declare_entry("theta", "0.5", Double(0), "theta");
prm.declare_entry("mesh_file", "mesh_file", Anything(), "Mesh File");
prm.declare_entry("refinement_mesh_degree",
                  "10",
                  Integer(0),
```

```
        "refinement_mesh_degree");  
prm.leave_section();  
  
prm.enter_section("Simulation parameters");  
simulation_time      = prm.get_double("simulation_time");  
time_step            = prm.get_double("time_step");  
theta                = prm.get_double("theta");  
mesh_file            = prm.get("mesh_file");  
refinement_mesh_degree = prm.get_integer("refinement_mesh_degree");  
prm.leave_section();
```

Una volta che tutte le impostazioni sono state lette correttamente vengono esposte da una struttura, chiamata **Settings** accessibile da tutti gli altri moduli all'interno del programma.

```
Settings settings;  
settings.get_parameters("parameter_file.prm");
```

5.2 Log e statistiche

Data la natura computazionalmente complessa dei programmi di CFD, una buona pratica per verificare se gli algoritmi o le strutture dati utilizzate durante lo sviluppo sono ottimali è quella di calcolare i tempi di esecuzione di varie istruzioni critiche all'interno del programma. È evidente quindi che bisogna costruire un sistema di logging di statistiche robusto e facile da usare, le librerie `std::chrono`, `dealii::logstream` e `dealii::logstream` ci semplificano il lavoro in quanto offrono molte funzionalità utili proprio per raccogliere informazioni e reindirizzarle sia su schermo che su file.

5.2.1 Implementazione

Il modulo di logging e statistiche è piuttosto semplice rispetto agli altri, infatti consiste principalmente di tre elementi:

- `std::clock`: classe che detta il tempo in base ai cicli della CPU, è estremamente preciso e viene usato per calcolare delta temporali tra istruzioni.
- `deallog`: rappresenta l'oggetto a cui passare le informazioni sottoforma di stringa attraverso l'operatore di inserzione «, che le riversa o su file o su console.
- `TimerOutput`: classe che viene usata per formattare le informazioni temporali di diversi tipi.

Inizializzando questi tre elementi appena dopo la creazione di **Settings** (5.1) rendiamo disponibili a tutto il programma, e per tutto il tempo di esecuzione, un punto di raccolta dei log (`deallog` formattati con `TimerOutput`) e un timer per le operazioni (`crono`).

```
std::ofstream log_file;
if(settings.write_to_file)
{
    log_file.open(settings.output_directory
                  + settings.output_filename
                  + ".log");
    deallog.attach(log_file);
    deallog.depth_file(1);
    statistics = std::
        make_unique<TimerOutput>(deallog.get_file_stream(),
                                TimerOutput::summary,
                                TimerOutput::cpu_times);
}
else
{
    deallog.depth_console(1);
    statistics = std::make_unique<TimerOutput>(deallog.get_console(),
```

```
TimerOutput::summary,  
TimerOutput::cpu_times);  
}
```

Utilizzabili in congiunzione attraverso il seguente esempio:

```
std::time_t start_time = std::chrono::system_clock::to_time_t(start);  
deallog << "StartTime:" << std::ctime(&start_time);
```

5.3 Interpretazione della Mesh

Come anticipato nel capitolo 3 la corretta interpretazione della mesh sulla quale operare è di fondamentale importanza quando si effettuano delle simulazioni CFD. Ancora una volta **dealii** ci viene in soccorso con alcuni strumenti. Tra questi abbiamo le classi **Triangulation**<dim,spacedim> e **GridIn**<dim,spacedim>. L'oggetto **Triangulation** offre diverse funzionalità di base per operare sulle celle o sui volumi di una mesh, rappresenta infatti la struttura responsabile di tenere traccia di quante e quali celle sono presenti sulla mesh in uso. **Triangulation** fa uso del paradigma 4.1.2.1 per gestire mesh di 1,2,3 dimensioni in spazi di 1,2,3 dimensioni. In base infatti ai parametri **dim** e **spacedim** è possibile realizzare simulazioni con diverse dimensioni senza dover stravolgere il codice e avendo sempre le migliori performance, infatti **dealii** ottimizza gli algoritmi di iterazione in base alle dimensioni su cui lavora.

La creazione di un oggetto **Triangulation** può avvenire in diversi modi, all'interno del progetto viene utilizzato l'oggetto **GridIn**. **GridIn** è una classe che implementa un meccanismo di creazione di triangolazioni in base a strutture a griglia. Riesce infatti a convertire file esportati in diversi formati in oggetti **Triangulation**, tra questi formati sono presenti: UCD (unstructured cell data), DB Mesh, XDA, Gmsh, Tecplot, NetCDF, UNV, VTK, ASSIMP, e Cubit. Chiaramente le mesh importate attraverso quest'oggetto non devono essere complicate e soprattutto non devono essere modificate attraverso meccanismi di suddivisione poiché **Triangulation** già può ridefinire la mesh aggiungendo più livelli di suddivisione.

5.3.1 Implementazione

Per raggiungere i risultati descritti nel capitolo 6 è stata utilizzata una mesh generata attraverso SALOME 4.4.2, che viene letta con l'ausilio di un oggetto **Mesher**, interpretandola attraverso gli strumenti precedentemente discussi. Dato un oggetto **Triangulation**<dim,spacedim> inizializzato, ma vuoto, si utilizza mesher per caricare all'interno della triangolazione la relativa mesh:

```
GridIn<dim> grid_in;  
grid_in.attach_triangulation(triangulation);  
std::string filename = settings.mesh_file;  
std::ifstream file(filename);  
grid_in.read_unv(file);
```

Ciò non basta se vogliamo preservare i `boundary_id` della mesh, è infatti necessario iterare tutte le cell e rimarcare le facce corrispondenti:

```
for (const auto &cell : triangulation.cell_iterators())  
    for (unsigned int face_number = 0;  
         face_number < GeometryInfo<dim>::faces_per_cell;  
         ++face_number)  
    {  
        if (cell->face(face_number)->at_boundary())  
            cell->face(face_number)  
                ->set_boundary_id(settings.walls_manifold);  
    }
```

è possibile marcare ulteriori boundaries attraverso lo stesso meccanismo, ma cambiando la logica che porta al `set_boundary_id()`. Per esempio se volessimo marcare un certo numero di facce all'interno di un intervallo di distanza dal centro della mesh possiamo usare il seguente metodo:

```
for (const auto &cell : triangulation.cell_iterators())
```

```

for (unsigned int face_number = 0;
    face_number < GeometryInfo<dim>::faces_per_cell;
    ++face_number)
{
    if(cell->face(face_number)->at_boundary())
    {
        const auto center = cell->face(face_number)->center();
        double radius = sqrt(pow(std::fabs(center(0)),2)
                               + pow(std::fabs(center(1)),2));
        if(center(2) > 0.015 - 1e-12 && radius > 0.005)
        {
            cell->face(face_number)
                ->set_boundary_id(settings.inlet_manifold);
        }
    }
}

```

Una volta caricata e marcata correttamente la mesh è possibile utilizzarla per tutte le computazioni necessarie all'interno della simulazione passando il riferimento all'oggetto **Triangulation** creato in precedenza.

5.4 Navier-Stokes

5.4.1 Inizializzazione

Avendo a disposizione sia le impostazioni della simulazione che la mesh si possono inizializzare correttamente i solver (3.3), partendo da quello responsabile della simulazione del flusso. La prima cosa da fare quando si inizializza il modulo Navier-Stokes è creare le strutture dati di supporto e impostare i gradi di libertà della simulazione (3.3.3). Per gestire i gradi di libertà di una simulazione ci avvaliamo della classe **DoF-Handler<dim>** disponibile in `deal.II/dofs/dof_handler.h` che data una mesh (o **Triangulation**) mette a disposizione una serie di funzionalità per iterare sui gradi di

libertà di tutti i vertici, celle, facce della stessa. Per ogni vertice, linea o quadrilatero questa classe salva una lista di indici di gradi di libertà presenti sull'oggetto. Il modulo utilizza due **DoFHandler**<dim>: uno per i gradi di libertà della velocità del flusso, uno per la pressione. Oltre ai gestori di DoF, in fase di inizializzazione del modulo, si impostano anche gli oggetti responsabili della gestione del sistema di elementi finiti e le formule gaussiane di quadratura.

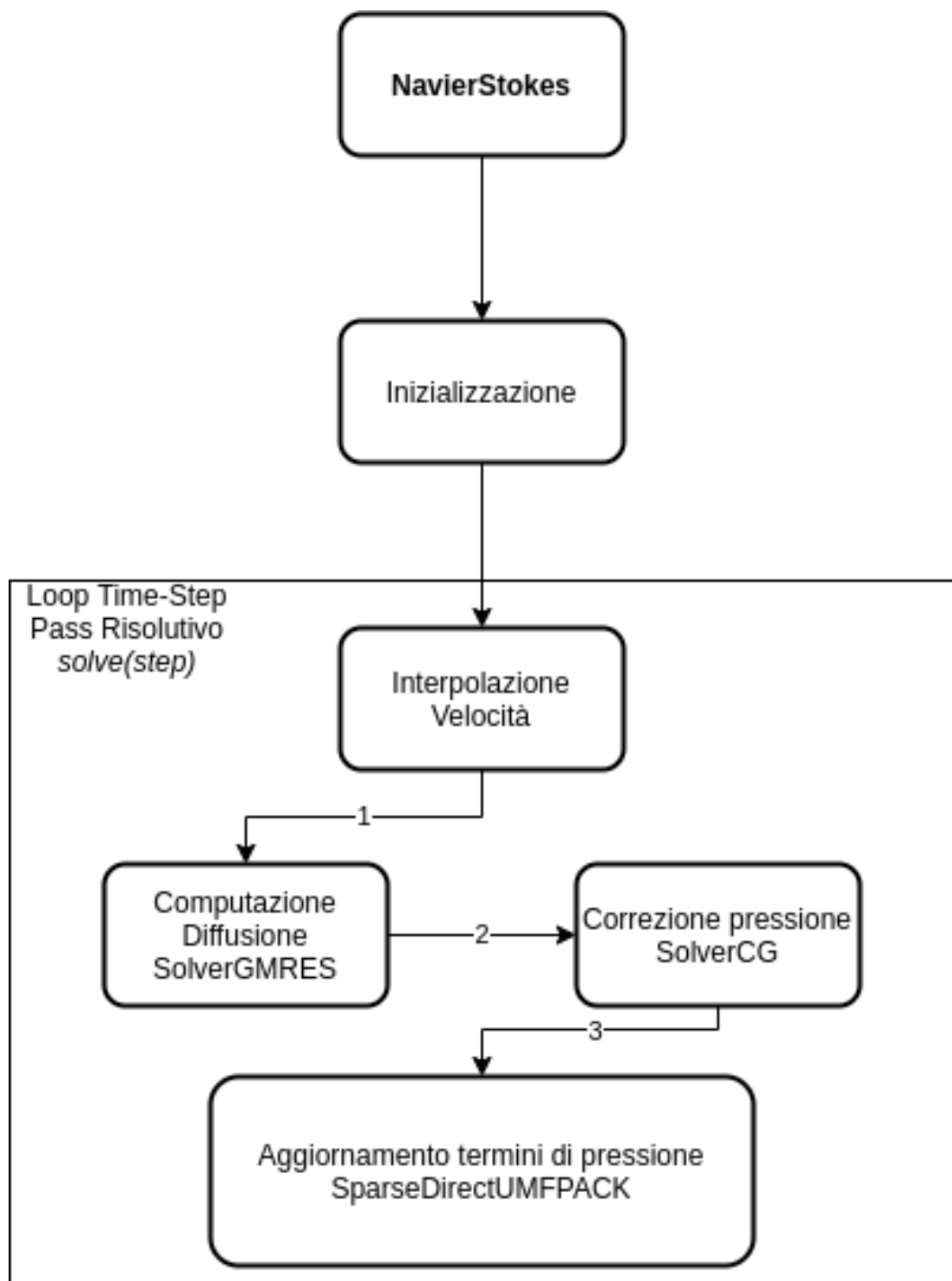


Figura 5.3: diagramma di flusso NavierStokes.

Una volta creato l'oggetto tramite:

```
NavierStokes navierstokes(settings, triangulation);
```

lo inizializziamo così:

```
navierstokes.setup_system();
```

5.4.2 Passo risolutivo

Dopo aver inizializzato correttamente il modulo è possibile richiamare la funzione: `solve(int step)` per risolvere un singolo time-step della simulazione. Vediamo in dettaglio come avviene la risoluzione del time-step e cosa comporta.

Il primo passo consiste nell'interpolare (3.3.2) le velocità, successivamente si effettuano i calcoli per la diffusione. L'interpolazione della velocità avviene per mezzo di oggetti **Function**<dim> messi a disposizione da dealii che ci permettono di computare in un dato punto una serie di valori in base a formule matematiche che impostiamo. Nel caso, per esempio, dell'immissione, all'interno dell'ugello, del flusso è stata creata una funzione denominata **Inlet Velocity**; che in base alla coordinata del punto in cui si calcola la velocità e una serie di impostazioni iniziali, computa il corretto valore che il flusso assume in termini di velocità. In una mesh possono agire tante funzioni interpolative quante ce ne sono bisogno. Una volta interpolati tutti i valori di velocità si passa a calcolare l'effettiva diffusione del flusso. Per procedere al calcolo diffusivo bisogna innanzitutto assemblare il sistema e applicare e interpolare le condizioni al contorno. Una volta calcolati i valori al contorno vengono applicati al sistema, che attraverso un solver GMRES (offerto sempre da dealii) computa la diffusione.

Dopo aver calcolato la diffusione del fluido bisogna proiettare e correggere per il termine di pressione. Per computare l'apporto dato dalla pressione al fluido ci avvaliamo di un Solver CG (**Conjugate Gradient**, o gradiente coniugato) che implementa metodi per la risoluzione di sistemi lineari di matrici reali simmetriche positive.

Ultimata la proiezione passiamo ad aggiornare le strutture dati che descrivono i valori di pressione utilizzando **SparseDirectUMFPACK**: un solver di sistemi lineari non

simmetrici. Concluso quest'ultimo passaggio il programma ha correttamente simulato il movimento del flusso all'interno della mesh utilizzando le equazioni di Navier-Stokes (2.2).

5.4.3 Sistemi non lineari

Il modulo Navier Stokes è stato implementato anche in una versione capace di risolvere sistemi di equazioni non lineari. Affinché il modulo riesca in questo intento è stato necessario introdurre uno schema risolutivo di tipo Newton-Raphson (2.4). Per utilizzare questa versione è necessario introdurre due ulteriori impostazioni ricavate sempre dall'oggetto **Settings**: il numero di iterazioni ammesse e la tolleranza per il metodo Newton-Raphson. Avendo inizializzato correttamente il modulo nella sua versione non lineare, si passa alla risoluzione dei time-step. Per ogni step viene assemblato il problema non lineare, si controllano eventuali errori residui e si passa al controllo della convergenza della soluzione. Una volta che abbiamo appurato errori e convergenza si assembla la tangente del sistema e si impongono le condizioni di Dirichlet. A questo punto abbiamo convertito un sistema non lineare in un sistema linearizzato che possiamo risolvere tradizionalmente.

5.5 Temperatura e Laser

Analizziamo ora i componenti che sono responsabili per la simulazione del trasferimento di calore all'interno del simulatore, fondamentali per studiare il comportamento del flusso di particelle e il comportamento dello stesso nell'intorno della superficie su cui depositare il materiale.

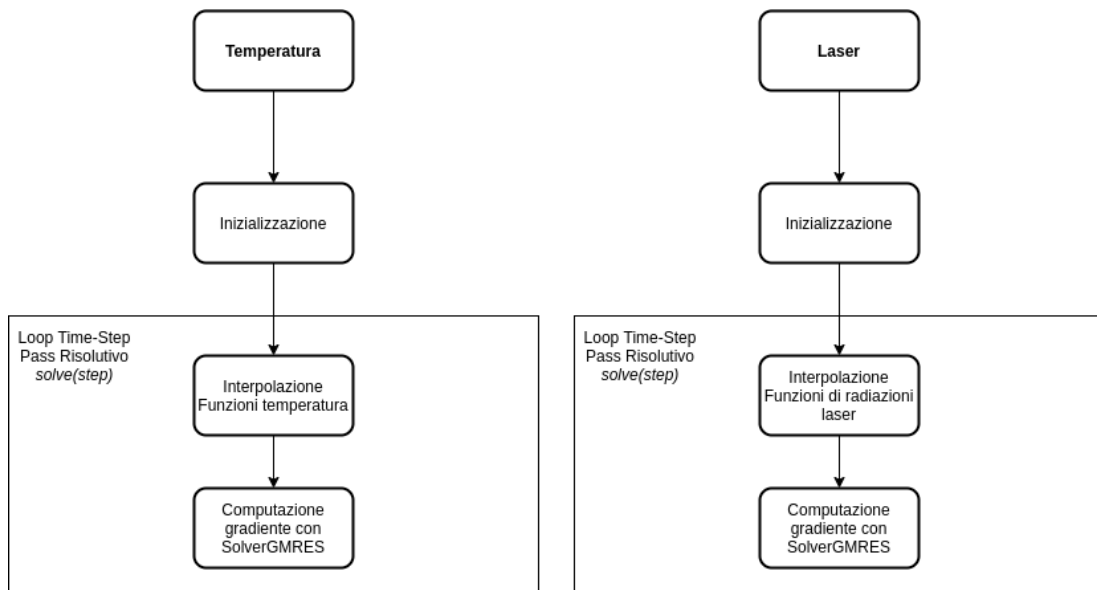


Figura 5.4: diagramma di flusso Temperatura e laser.

L'inizializzazione dei moduli è simile a quella che abbiamo già effettuato per 5.4:

```
Temperature temperature(settings,triangulation);
temperature.setup_system();
```

```
Laser laser(settings,triangulation);
laser.setup_system();
```

di seguito analizziamo nel dettaglio le differenze fra i due moduli.

5.5.1 Temperatura

L'inizializzazione del modulo temperatura comporta la valorizzazione, attraverso l'oggetto **Settings**, di diverse variabili necessarie a regolare il comportamento del modulo. Riportiamo di seguito alcune variabili particolarmente rilevanti:

- Tempo di applicazione dell'effetto **Plateu**
- Temperatura massima e minima

- Temperatura in prossimità dell'ingresso dell'ugello

Successivamente alla valorizzazione delle variabili, vengono inizializzate le strutture dati relative ai gradi di libertà del modulo temperatura, e i relativi vincoli. Il modulo temperatura prevede una correzione in fase di risoluzione del time-step che necessita di strutture sincronizzate con time-step precedenti, questo implica la creazione e la gestione di vettori contenenti informazioni "in ritardo" rispetto alla simulazione di uno e due time-step. Il passo finale dell'inizializzazione riguarda la prima interpolazione delle strutture contenenti i gradi di libertà con le rispettive condizioni iniziali. Risulta molto simile anche il resto del funzionamento del modulo temperatura rispetto a 5.4, infatti una volta inizializzato il componente si passa alla computazione dello step corrente, sempre attraverso una funzione `solve(step)`. Il solver utilizzato in questo modulo rimane **SolverGMRES** come in 5.4. La differenza con il modulo precedente sta ovviamente nel tipo di condizioni al contorno e nel tipo di funzioni che vengono computate per ogni punto. Vengono infatti interpolati i valori di temperatura del flusso e non più di velocità, per ogni cella della mesh. Per esempio nel caso dell'oggetto preso in esame si computa una zona di alta temperatura presso la parte iniziale del condotto dell'ugello, che nel corso della simulazione influenza il flusso passante per esso. Il modulo temperatura è intrinsecamente collegato a quello del laser 5.5.2 e entrambi, congiuntamente a 5.4, influenzano le proprietà delle particelle (5.6).

5.5.2 Laser

Questo modulo si occupa di simulare l'effetto di un fascio laser all'interno dell'ugello, che scalda il flusso e le particelle che lo attraversano influenzandone il comportamento. L'inizializzazione del modulo è simile a quella che abbiamo già effettuato per 5.4 e 5.5.1, in questo caso, però, necessitiamo di solo due proprietà da **Settings**:

- La potenza del fascio laser
- Il raggio del fascio laser

inoltre sono presenti meno strutture dati. In particolare non avendo bisogno di risultati antecedenti al passo risolutivo corrente non ci sono strutture che referenziano stati

precedenti come nel modulo temperatura. Prima di terminare l’inizializzazione, come di consueto ormai, interpoliamo le **Function**<dim> specifiche per il modulo laser con la struttura che gestisce i gradi di libertà (**DoFHandler**). Successivamente in fase di computazione del time-step ci avvaliamo un’altra volta di un **SolverGMRES** come in 5.4 e 5.5.1.

5.6 Particelle

Il modulo che gestisce le particelle è stato realizzato sulla base di una struttura offerta dalla libreria dealii 4.2.5, **Particle**<dim,spacedim> che analizzeremo di seguito. La classe Particle rappresenta una particella singola all’interno di un dominio caratterizzato da un oggetto **Triangulation**<dim,spacedim>. Particle agisce come struttura dati per una singola particella al cui interno troviamo diverse informazioni fondamentali per la corretta simulazione della stessa, tra queste informazioni riportiamo di seguito le più utilizzate all’interno del progetto:

- Indice della particella (un identificativo univoco all’interno della simulazione)
- Posizione della particella nel sistema di riferimento della dominio intero
- Posizione della particella nel sistema di riferimento della cella in cui risiede
- Un vettore di proprietà personalizzabili in base alle necessità della simulazione. In questo progetto il vettore viene utilizzato per salvare informazioni quali velocità e temperatura della particella.

Analogamente a quanto utilizzato per gestire i gradi di libertà di 5.4, 5.5.1 e 5.5.2, dealii offre un oggetto in grado di manipolare in maniera controllata e precisa le particelle di una simulazione: **ParticleHandler**<dim,spacedim>. La classe ParticleHandler infatti espone diversi accessori per iterare particelle in un particolare dominio.

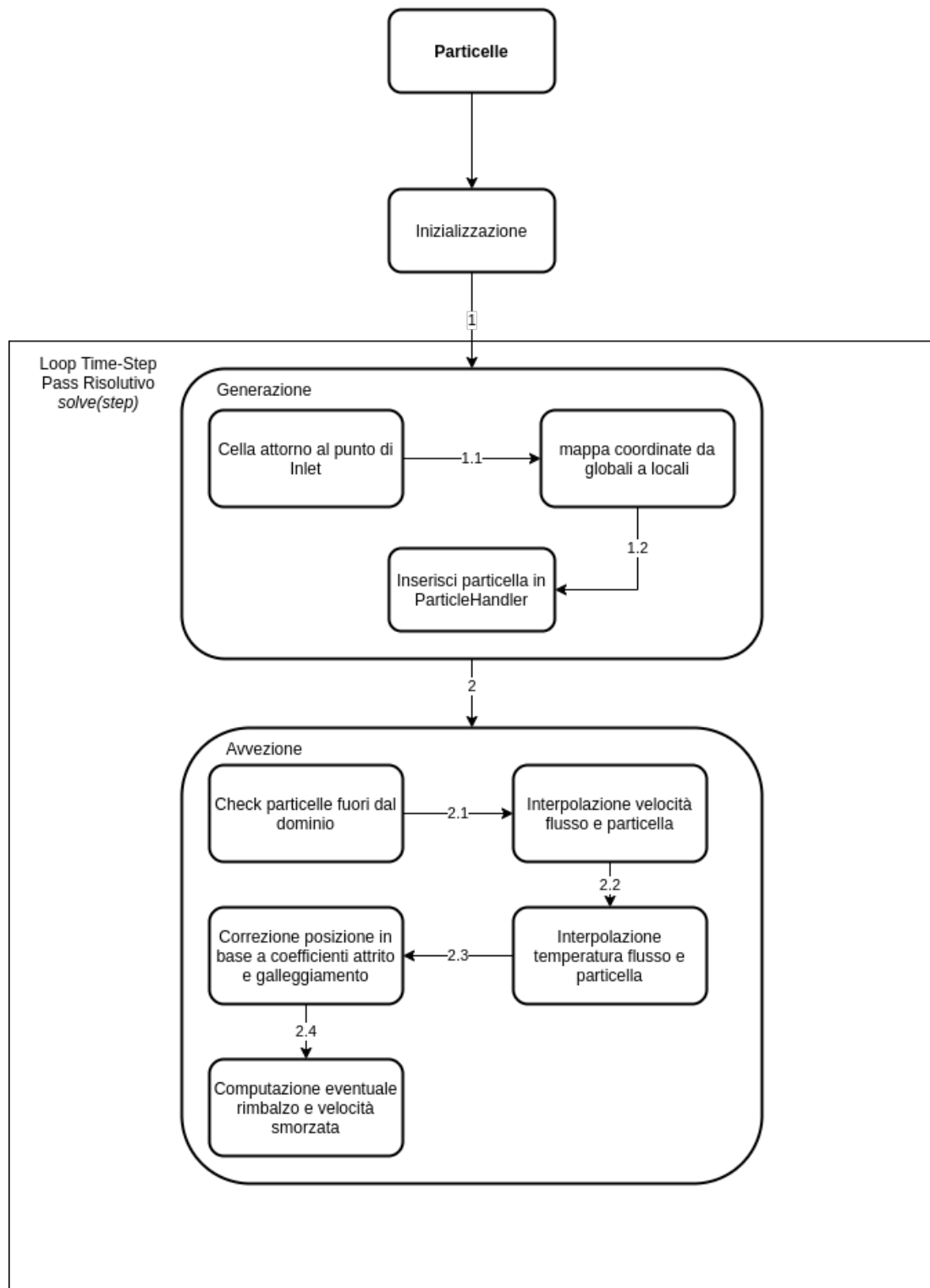


Figura 5.5: diagramma di flusso Temperatura e laser.

Ovviamente anche in questo modulo vi è una fase di inizializzazione eseguita però diversamente dagli altri:

```
ParticlesDynamic particles_dynamic(settings,  
                                   navierstokes,  
                                   temperature,  
                                   laser,  
                                   triangulation);  
  
particles_dynamic.generate_particles();
```

Infatti notiamo che quando creiamo l'oggetto principale del modulo, **ParticlesDynamic**, passiamo un riferimento agli altri moduli creati in precedenza. Questo perché, come anticipato, le particelle sono fortemente influenzate dai risultati dei moduli precedenti. Le impostazioni (5.1) che vengono passate a questo modulo riguardano i principali aspetti che si vogliono parametrizzare all'interno di una simulazione di particelle, per la singola particella. Di seguito è riportata una lista delle più importanti:

- Massa della particella
- Durata della generazione delle particelle come flusso in entrata (per quanto tempo generiamo particelle)
- Densità della particella
- Diametro della particella
- Temperatura iniziale delle particelle
- Calore specifico
- Rateo di assorbimento calore
- Coefficiente di convezione

Inoltre al posto della funzione `setup_system()` troviamo `generate_particles()` che, come si evince dal nome scelto, è responsabile della generazione delle particelle all'interno della simulazione. Analizziamo ora le due principali funzionalità che offre il modulo di particelle: la generazione e l'avvezione delle stesse.

5.6.1 Generazione

Per rendere facilmente personalizzabile il processo di generazione particelle è stato realizzato un sottomodulo capace di interpretare determinate celle marcate come sorgenti di particelle. Questo permette di utilizzare diverse mesh e diverse sorgenti senza il bisogno di dover modificare il programma. Nel caso preso in esame, la generazione avviene in prossimità dell'ugello secondo i parametri passati in fase di inizializzazione.

```
auto ref_cell = GridTools::find_active_cell_around_point(
    dof_handler_temperature,
    location);

Point<dim> ref_p = StaticMappingQ1<dim>::mapping
    .transform_real_to_unit_cell(ref_cell, location);

auto it = particle_handler.insert_particle(Particles::Particle<dim>(
    location,
    ref_p,
    next_particle_index),
    ref_cell);
```

Notiamo come dopo aver calcolato, secondo i criteri selezionati, il punto dove collocare la particella (`location`), dobbiamo effettuare alcune trasformazioni prima di crearla. In particolare occorre prima di tutto trovare la cella attorno al punto scelto attraverso l'accessore **GridTools**, mappare le coordinate sul sistema di riferimento della cella trovata e infine inserire attraverso il `ParticleHandler` la particella. In questa fase nella

particella inizializziamo anche il vettore di proprietà con la temperatura iniziale ricavata attraverso le impostazioni.

5.6.2 Avvezione

L'avvezione delle particelle avviene nello stesso ciclo discretizzato degli altri moduli, dopo aver completato tutti i relativi calcoli dell'iterazione corrente. Il procedimento iniziale per procedere al calcolo del movimento delle particelle consiste in un iterazione attraverso tutte le particelle contenute nel **ParticleHandler**:

```
for (auto p = particle_handler.begin();
     p != particle_handler.end();
     p++)
```

all'interno di questo ciclo, in ordine, controlliamo se alcune particelle sono uscite dai confini del dominio in un iterazione precedente e le eliminiamo. Questo controllo è possibile sempre grazie a `find_active_cell_around_point()` che genera un'eccezione in caso la particella non sia all'interno di una cella. Successivamente, per ogni modulo precedentemente descritto, ricaviamo dalla cella in cui si trova la particella informazioni riguardanti velocità del flusso, temperatura del flusso e potenza del fascio laser. Dopo aver correttamente recuperato queste informazioni possiamo aggiornare le proprietà della particella e procedere al calcolo della nuova posizione della stessa. Il calcolo della nuova posizione inizia con il recupero della velocità della particella allo step precedente, attraverso le proprietà precedentemente citate. Una volta recuperato lo stato precedente iniziamo l'interpolazione tra la velocità precedente della particella e la velocità del flusso nella cella dove risiede la particella. Ogni interpolazione avviene per ogni componente (x, y, z nel nostro caso). Una volta terminata l'interpolazione con la velocità del flusso consideriamo il coefficiente d'attrito dinamico e il fattore di galleggiamento:

```
auto drag_force = 0.5
    * drag_coefficient
    * fluid_density
    * particle_area
```

```
        * relative_velocity;

auto buoyancy = vector_gravity * particle_density
               * particle_volume
               * ((particle_density - fluid_density)
                 / particle_density);
```

che inevitabilmente influenzeranno la velocità calcolata appena prima. Successivamente si introducono i calcoli per aggiornare la temperatura della particella, in particolare si considera sia il fascio laser che la temperatura del fluido. In termini pratici calcoliamo quanto calore viene assorbito dalla singola particella da entrambi i fattori:

```
auto laser_power_absorption = 0.25
                             * particle_absorptivity_rate
                             * laser_field;

auto particle_fluid_convection = particle_convection_coefficient *
    (p->get_properties()[num_temperature]
    - temperature_interpolated_field)
    *particle_area;

auto new_particle_temperature =
    p->get_properties()[num_temperature]
    + ((dt / (particle_mass * particle_specific_heat)) *
      (laser_power_absorption - particle_fluid_convection));
```

Terminate queste computazioni abbiamo valori aggiornati per temperatura e velocità e possiamo procedere al calcolo della nuova posizione della particella. Banalmente possiamo calcolare la nuova posizione aggiungendo a quella corrente la nuova velocità moltiplicata per il delta temporale che intercorre fra due time-step:

```
Point<dim> new_location = current_location + dt * new_particle_velocity;
```

Prima di confermare la posizione e assegnarla alla particella, però, dobbiamo controllare la presenza di eventuali rimbalzi o collisioni. Per fare ciò controlliamo se la nuova location risiede sempre all'interno della mesh, se così non fosse allora dobbiamo procedere a calcolare il rimbalzo della particella. La collisione, e il relativo rimbalzo, viene calcolato iterando tutti i lati della mesh considerati come "muri" (superfici che non possono essere attraversate, opportunamente marcate) e controllando, per ognuno di essi, se la particella si trova in prossimità di una delle facce del muro. In caso di esito positivo si procede a calcolare se il segmento descritto dalle posizioni (vecchia e nuova) interseca la superficie, o faccia, del muro. Se il segmento effettivamente interseca la superficie si procede con il calcolo del vettore di rimbalzo e, quindi, la corretta posizione della particella. Il rimbalzo oltre ad influenzare la direzione è responsabile anche di una diminuzione di velocità della particella, regolata dalle leggi degli urti. Dopo aver calcolato anche l'eventuale rimbalzo possiamo finalmente assegnare alla particella la nuova posizione e i nuovi valori di temperatura e velocità. Questi calcoli, come già accennato, vengono ripetuti per ogni particella, per ogni step di simulazione; appare evidente quindi che al crescere del numero delle particelle e dei gradi di libertà degli altri moduli, la simulazione diventa sempre più onerosa in termini di risorse computazionali.

5.7 Parallelizzazione

I moduli 5.4 e 5.5 utilizzano gli oggetti del namespace dealii `WorkStream` [TKB16] per operare su più thread (o più processori). `WorkStream` si basa interamente su 4.2.1 per bilanciare il carico di lavoro tra thread e su MPI per parallelizzare il lavoro su più calcolatori, condividendo informazioni e memoria. `WorkStream` rende disponibile a ogni calcolatore una copia dei `DoFHandler` in locale, mentre per matrici globali e vettori di soluzione condivide solo una parte (relativa ai propri calcoli). Una volta che il calcolatore ha a disposizione tutto il necessario esegue le computazioni necessarie e restituisce il risultato. Se un thread finisce una computazione con largo anticipo allora gli algoritmi di bilanciamento di TBB reindirizzeranno il carico di task sul thread scarico. Di seguito, un esempio di parallelizzazione utilizzando `WorkStream`:

```
WorkStream::run(
```

```

IteratorPair(IteratorTuple(ns.dof_handler_velocity.begin_active(),
                           dof_handler_temperature.begin_active())),
IteratorPair(IteratorTuple(ns.dof_handler_velocity.end(),
                           dof_handler_temperature.end())),

cell_worker_active,

copier_active,

T_ScratchData(*this, ns),

T_CopyData(*this));

```

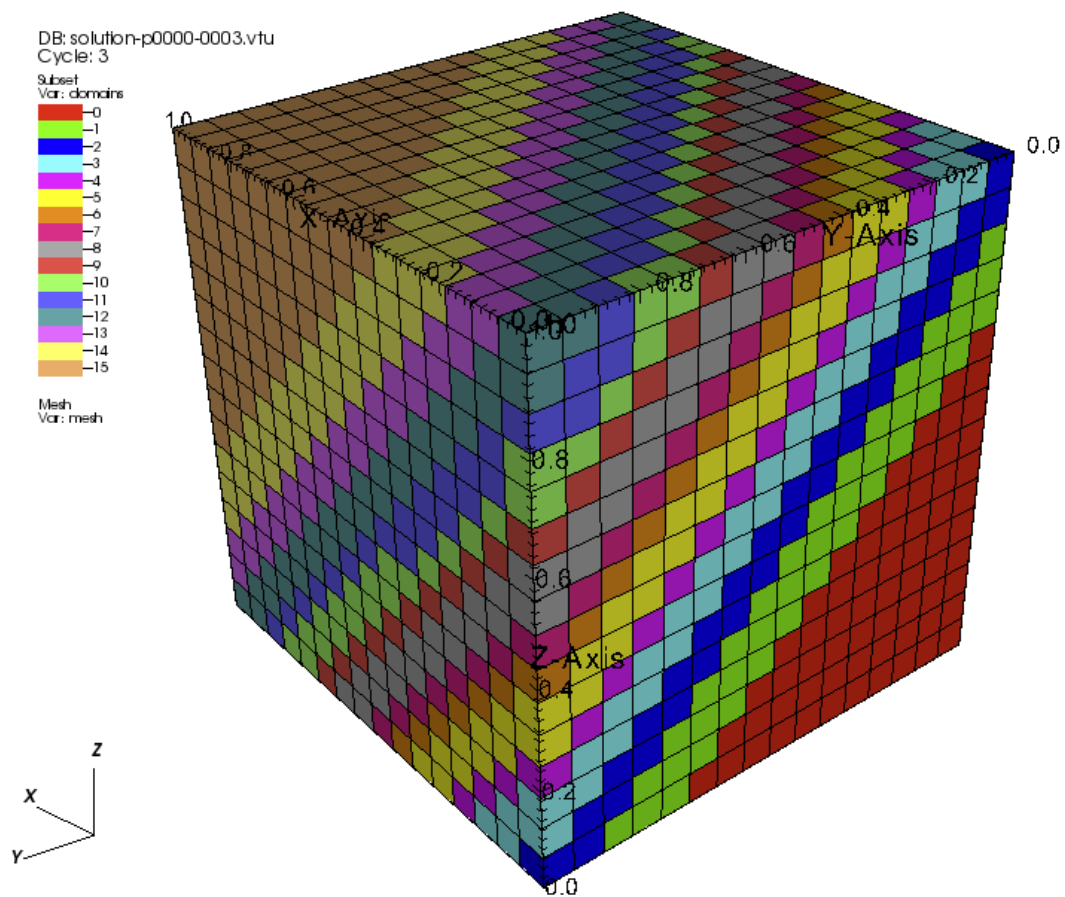


Figura 5.6: Esempio di mesh distribuita su più thread, differenziati per colore.

Con dealii è possibile anche distribuire gli oggetti Triangulation, entreremo nel dettaglio

di questo particolare nella sezione Conclusioni e sviluppi futuri (6.4).

5.8 Output

Dopo aver effettuato i calcoli di tutti i moduli è necessario, per ogni step, esportare i dati computati per renderli fruibili in programmi di postprocessing e visualizzazione (4.4). Per rendere il processo di output il più omogeneo possibile, ogni modulo implementa una funzione di output chiamata `output_results()` che gestisce in maniera autonoma la creazione dei file di output secondo i formati scelti per ogni modulo. Tutti i moduli tranne quello di simulazione particelle utilizzano un oggetto condiviso per generare un file con estensione VTK, facilmente importabile in **Paraview**(4.4.1). L'oggetto utilizzato, ancora una volta offerto da dealii (4.2.5), è **DataOut**<dim> che permette di esportare le strutture **DoFHandler** utilizzate durante le simulazioni. DataOut offre una funzione denominata `build_patches()` che itera su tutte le celle di una mesh (o Triangulation) salvate all'interno di un DoFHandler, precedentemente aggiunto all'oggetto DataOut, salvando tutte le informazioni di output in memoria. Una volta che sono state generate le informazioni di output è possibile invocare la funzione `write()` di DataOut, indicando un formato, per salvare su file ciò che è stato generato con `build_patches()`. Questo approccio rende possibile salvare in più formati gli stessi dati in maniera facile e veloce.

Navier-Stokes All'interno del modulo Navier-Stokes la funzione di output aggiunge i seguenti DoFHandler all'oggetto DataOut:

```
data_out.add_data_vector(dof_handler_velocity,
                        u_n,
                        "velocity",
                        stokes_component_interpretation);
data_out.add_data_vector(dof_handler_pressure, pres_n, "pressure");
```

Si evince dal listato che il modulo Navier-Stokes fornisce in output informazioni riguardo la velocità e la pressione del fluido. Una volta passati entrambi gli oggetti a DataOut possiamo richiamare `build_patches()` e passare al prossimo modulo.

Temperatura Analogamente a quanto fatto per il modulo precedente, e con le stesse istruzioni, aggiungiamo a DataOut il DoFHandler del modulo temperatura e costruiamo i dati di output.

Laser Identicamente al modulo temperatura, alleghiamo a DataOut le strutture riguardanti il fascio laser all'interno dell'ugello.

Particelle Essendo la struttura dei dati delle particelle completamente differente da quella degli altri moduli, anche il processo di output è sostanzialmente diverso. Il formato del file prodotto dal modulo particelle è un **CSV (comma separated values)**, poiché si tratta di generare una lista di particelle e valori associati. Per ogni step viene generato un file CSV denominato in base al numero di step corrente (al fine di poter importare gruppi di file in Paraview) al cui interno disponiamo i dati delle particelle, in modo tale che ogni particella occupi una riga del file. In una singola riga del file troviamo tutte le informazioni necessarie all'analisi della simulazione delle particelle:

- Posizione (divisa per componenti x, y, z)
- Temperatura
- Temperatura Fluido
- Velocità (divisa per componenti x, y, z)
- Velocità fluido (divisa per componenti x, y, z)
- Potenza Laser
- Identificatore Particella

Tutti i file prodotti, due per ogni step considerando che le particelle scrivono su un file separato dal resto dei moduli, vengono salvati all'interno di una cartella di output precedentemente specificata nel file di parametri (5.1). I file vengono denominati in modo tale che poi possano essere importati in Paraview in gruppi, ovvero seguendo questo pattern `nomeFile_numeroStep.estensioneFile`, per esempio per i moduli che esportano VTK abbiamo `output_0001.vtk`.

Capitolo 6

Risultati

In questo capitolo verranno analizzati i risultati prodotti dal simulatore realizzato in questo progetto di tesi. Durante lo sviluppo di questo simulatore sono stati svolti molti esperimenti relativi al funzionamento dei vari moduli e all'interazione tra flusso e particelle, di seguito, per ovvie ragioni, mostreremo solo quelli effettuati nello stato più avanzato di sviluppo. In particolare verranno analizzati i risultati dei vari moduli in maniera separata per poi combinarli in unico risultato sperimentale. Per assemblare le figure riportate è stato utilizzato Paraview (4.4.1) utilizzando il modulo di RayTracing (4.4.1.1).

6.1 Mesh

La mesh è stata generata con Salome (4.4.2) ed è composta da due coni concentrici rivolti verso il basso. Nel cono più esterno passa il fluido e le particelle, mentre in quello interno il fascio laser. Di seguito una serie di immagini che descrivono al meglio la mesh, notare la disposizione dei vertici e l'aumento di risoluzione in prossimità dell'uscita dell'ugello.

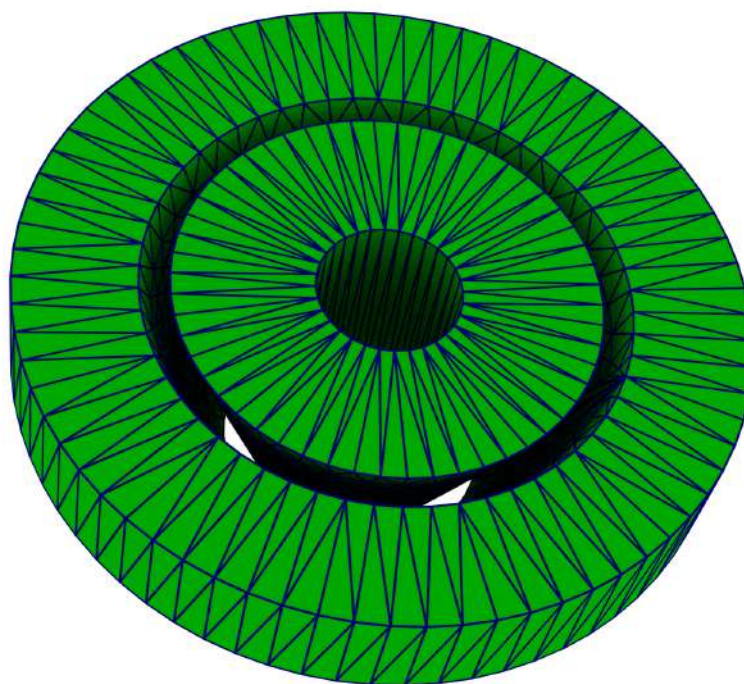


Figura 6.1: Ugello visto dall'alto.

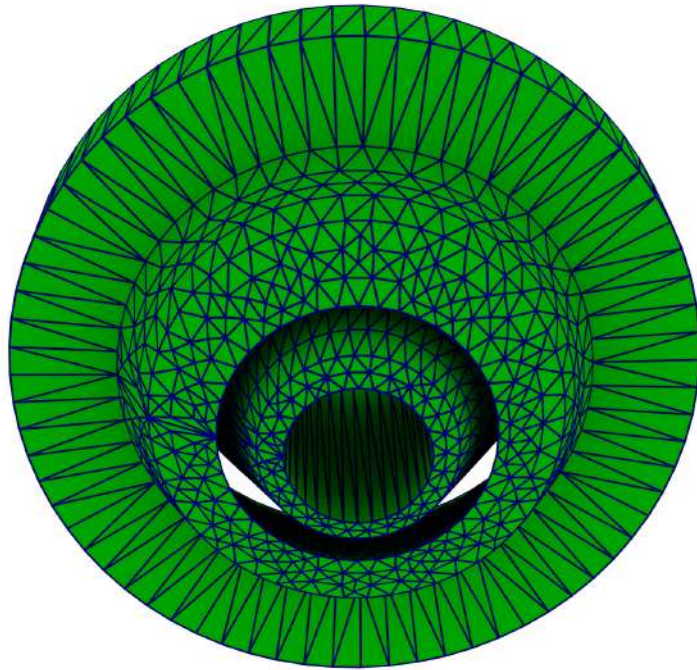


Figura 6.2: Ugello visto dal basso.

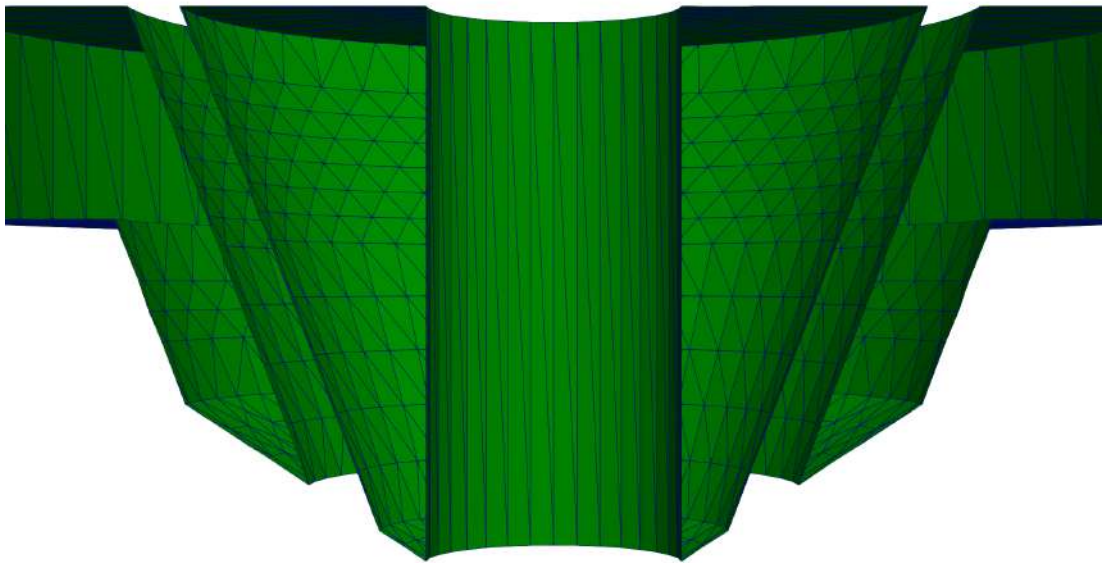


Figura 6.3: Ugello sezione laterale.

6.2 Fluido

Analizzando l'immagine seguente notiamo una sezione dell'ugello in verde e il flusso di gas all'interno dello stesso in gradazioni di blu e rosso, in base alla velocità assunta. Tralasciando gli errori di visualizzazione e compenetrazione dovuti a come Paraview visualizza gli oggetti sovrapposti, notiamo che il fluido viene accelerato nei condotti dell'ugello e acquista velocità in fase di uscita. Questo comportamento è supportato dal

fatto che la pressione è maggiore nella parte più alta dell'ugello.

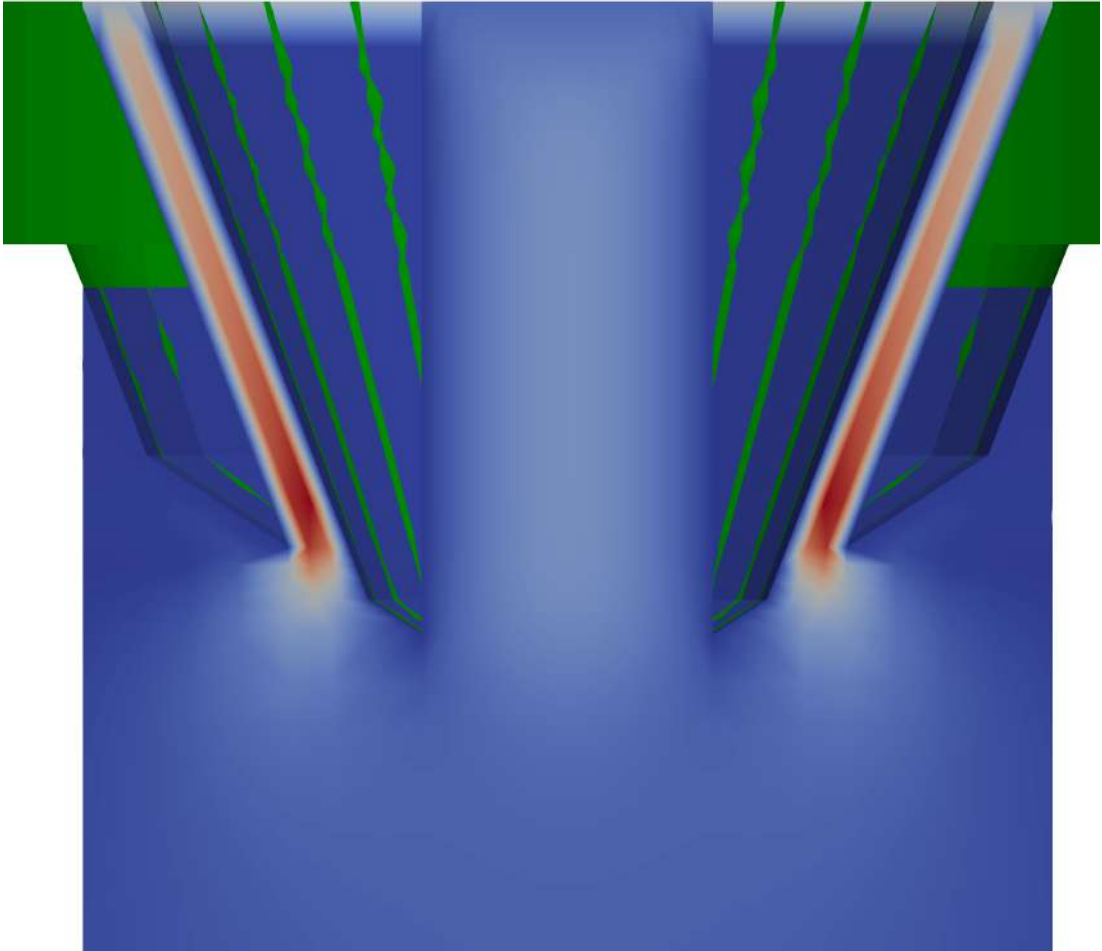


Figura 6.4: Dettaglio flusso in entrata ugello, sezione laterale. Velocità fluido.

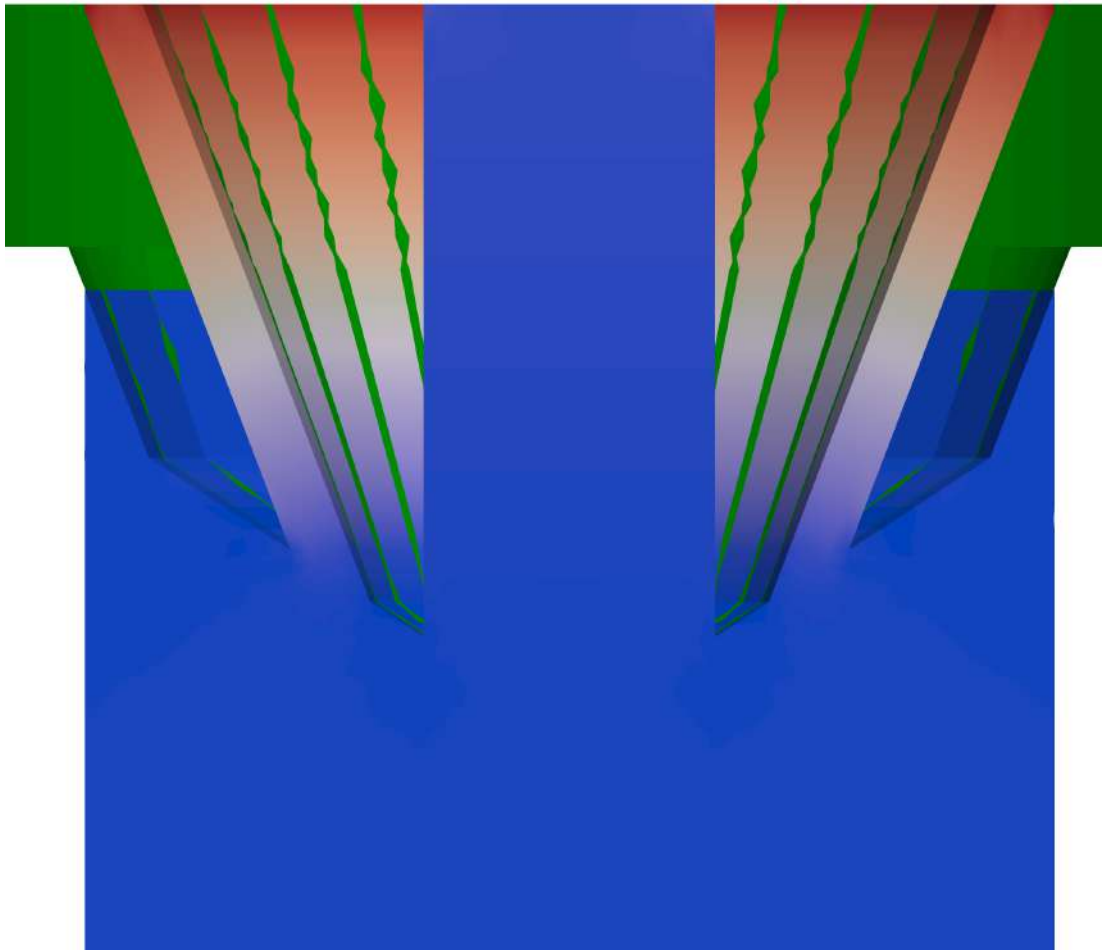


Figura 6.5: Dettaglio flusso in entrata ugello, sezione laterale. Pressione fluido.

Per realizzare il gruppo di immagini precedente è stato importato il gruppo di file VTK prodotto dal solutore all'interno di Paraview. Una volta importato è stato applicato un filtro di tipo **Clip** che ci permette di sezionare la mesh rispetto a un'asse. In questo caso il clip è stato aggiunto in prossimità del centro lungo l'asse verticale. Il gradiente di velocità e pressione è stato impostato automaticamente da paraview in quanto nel file generato sono stati inclusi i rispettivi campi vettoriali (come visto nella sezione 5.8). Infine come ultimo passaggio è stata attivata la vista ortografica in due dimensioni per

eliminare distorsioni nel render.

6.3 Laser

Il laser viene simulato come un cilindro, che passa per il vertice del cono centrale, al cui interno si registra una temperatura più alta rispetto al resto del volume. Nell'immagine sottostante è possibile notare come il fascio si propaga per tutto il volume e non intersechi l'ugello.

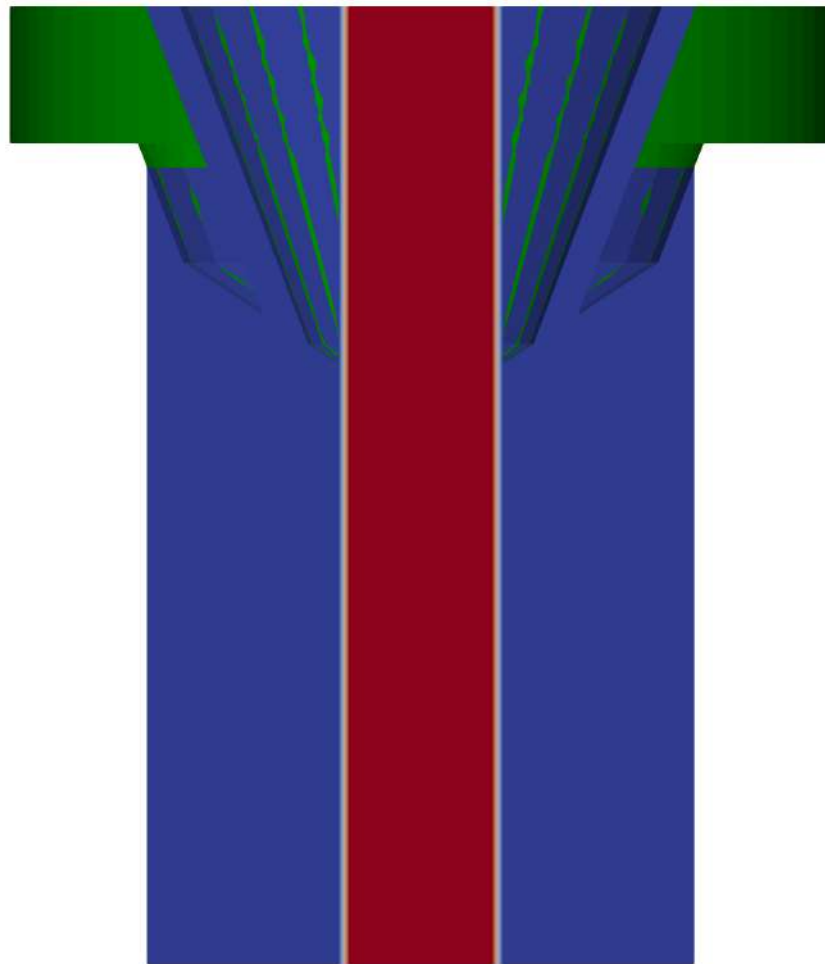


Figura 6.6: Ugello con radiazione laser, sezione laterale.

Il procedimento per produrre quest'immagine è identico a quello precedente, con l'accortezza, però, di attivare il gradiente per il campo scalare relativo alla radiazione laser.

6.4 Particelle

Analizziamo ora in maniera dettagliata la generazione delle particelle e la loro simulazione. Nella seguente immagine è possibile valutare, tramite una vista dall'alto della soluzione, l'anello di particelle che si genera nella fase iniziale della simulazione. Per importare il file **CSV** all'interno di Paraview è stato utilizzato un filtro chiamato "TableToPoints", che, dato in input proprio un file **CSV**, permette di mappare ogni riga del file a punti nello spazio tridimensionale. Per rendere più visibili le particelle è stato omesso il fluido e l'ugello da questi render.

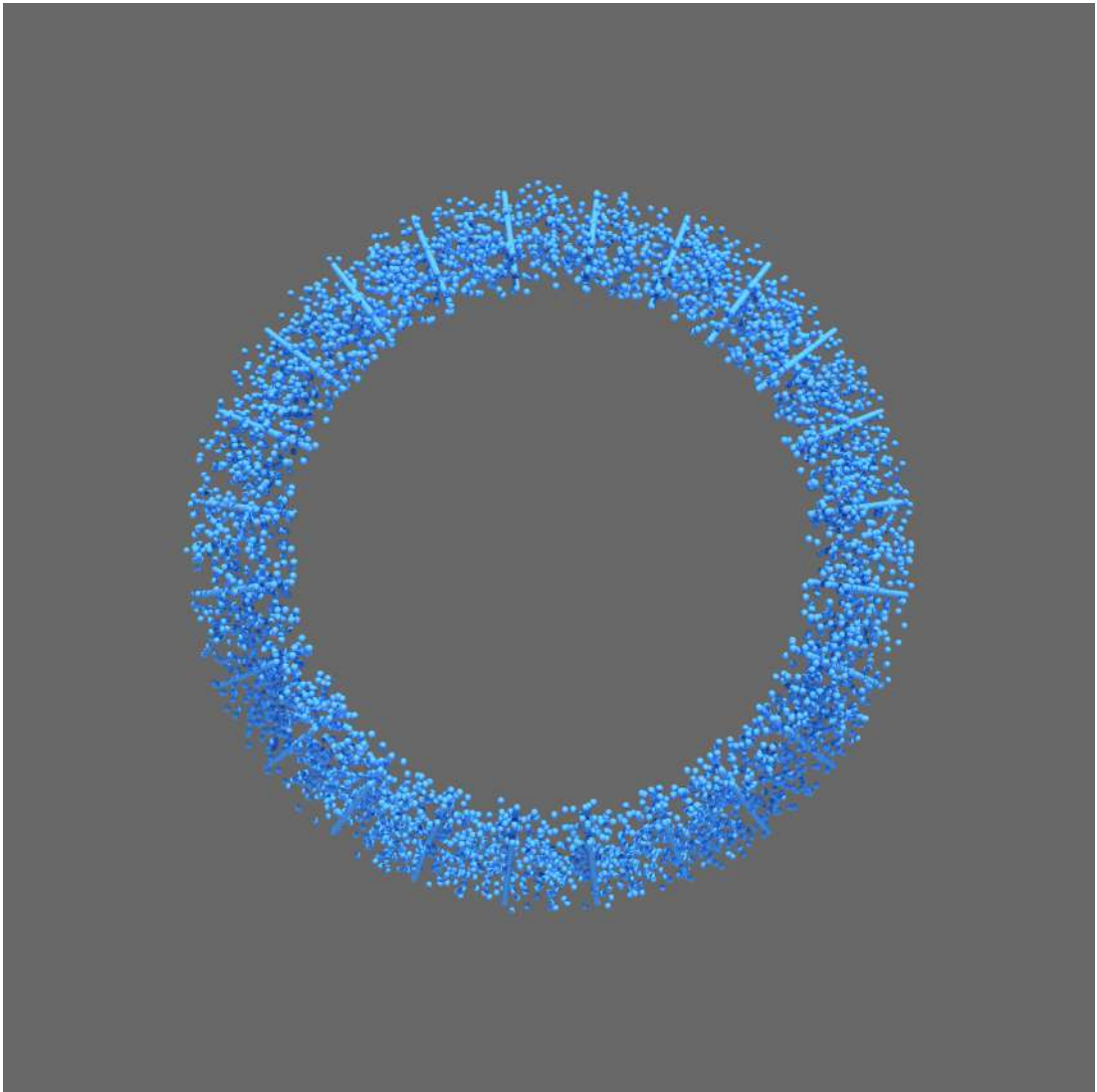


Figura 6.7: Anello di particelle in entrata, vista dall'alto.

Come accennato in precedenza (5.6) le particelle possiedono un campo scalare che rappresenta la temperatura. Questo campo è univoco per particella e ci permette di applicare un gradiente basato su di esso all'interno di Paraview. Intuitivamente il colore blu rappresenta particelle fredde, quello rosso calde.

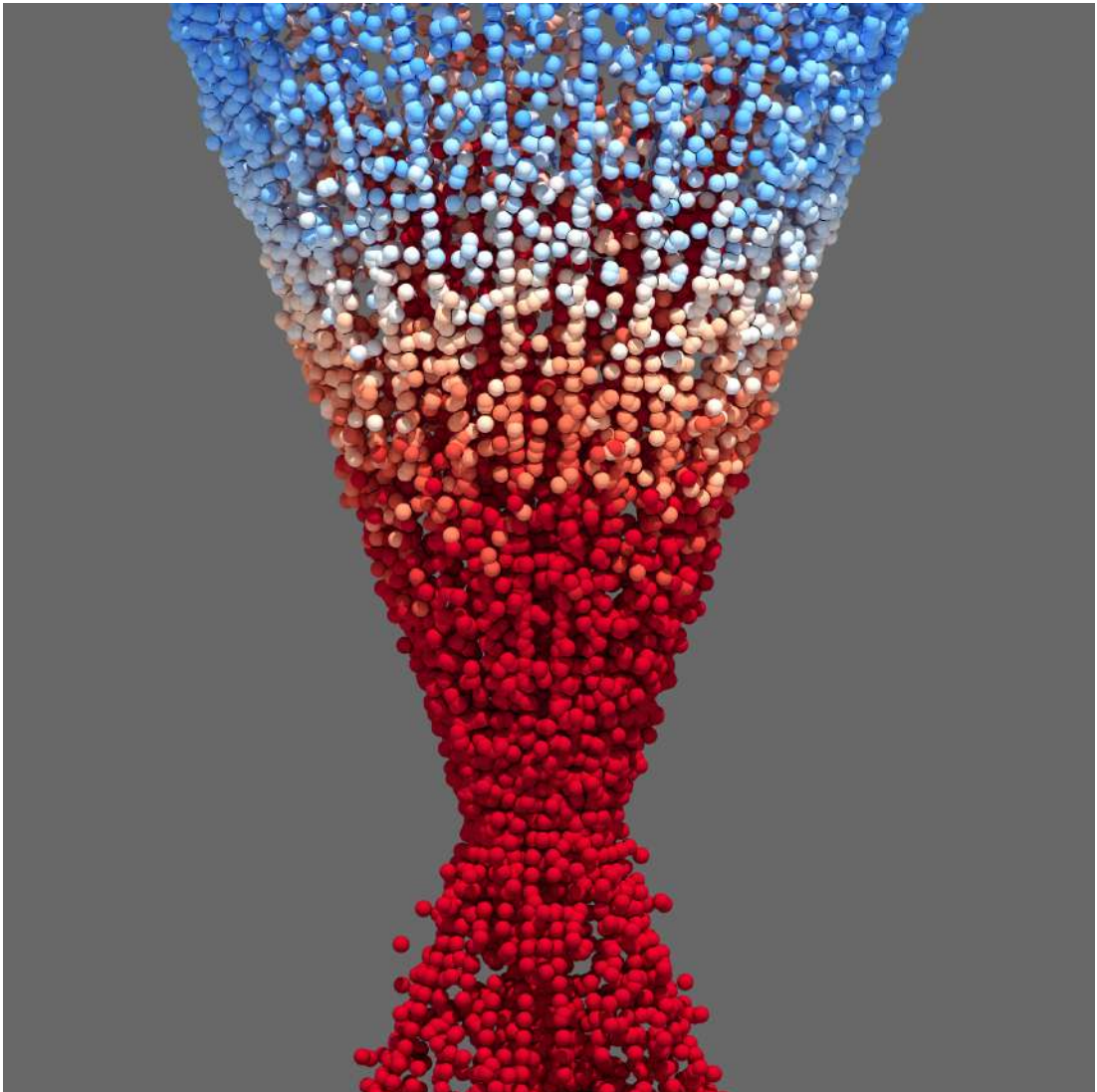


Figura 6.8: Particelle con colorazione basata su valori temperatura, vista laterale.

Notare come le particelle cambino temperatura proprio in corrispondenza del fascio laser.

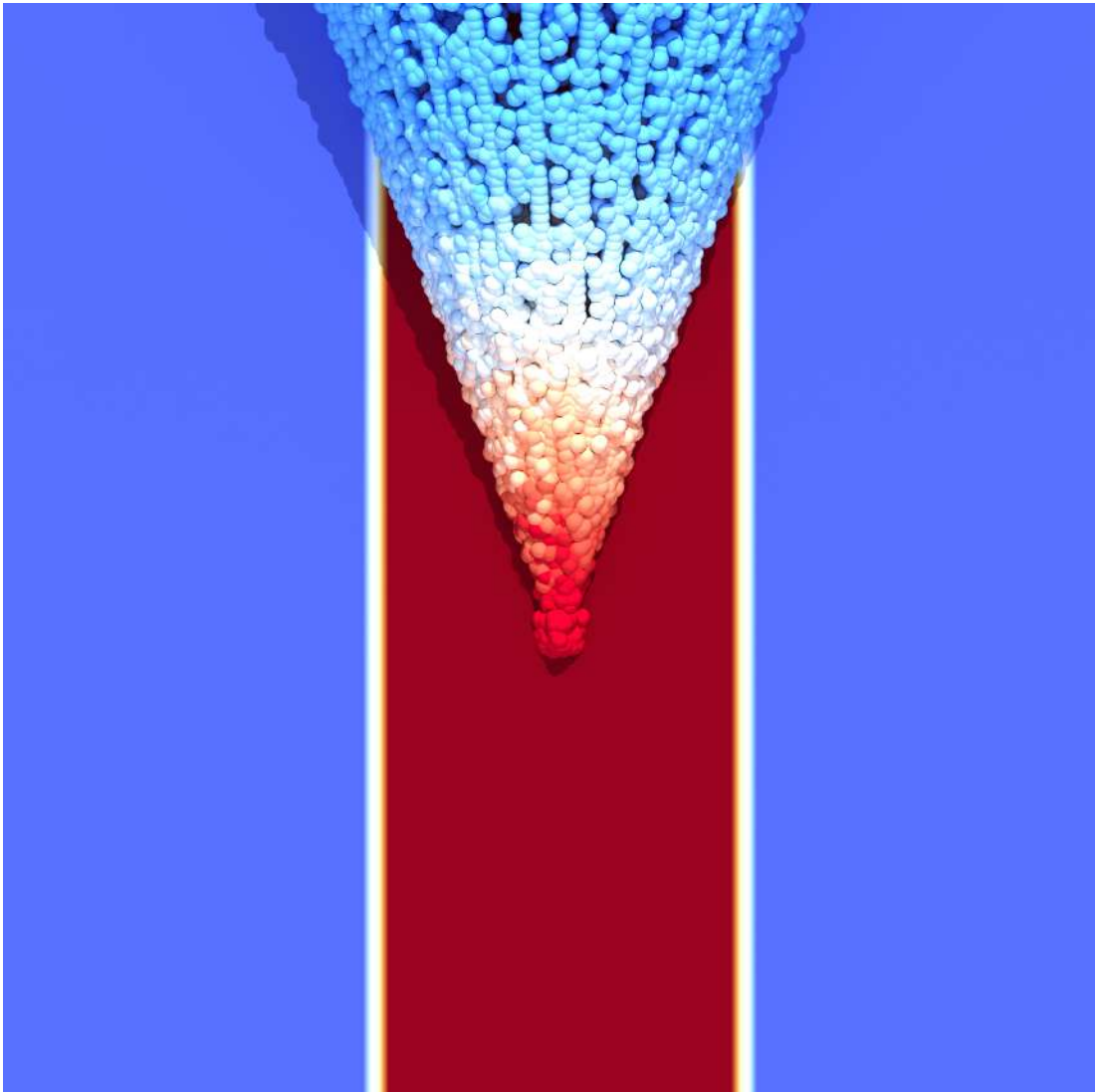


Figura 6.9: Dettaglio particelle riscaldate da laser, sezione laterale.

Nei seguenti render è possibile vedere una sezione della completa simulazione, con ugello, velocità del flusso e particelle.

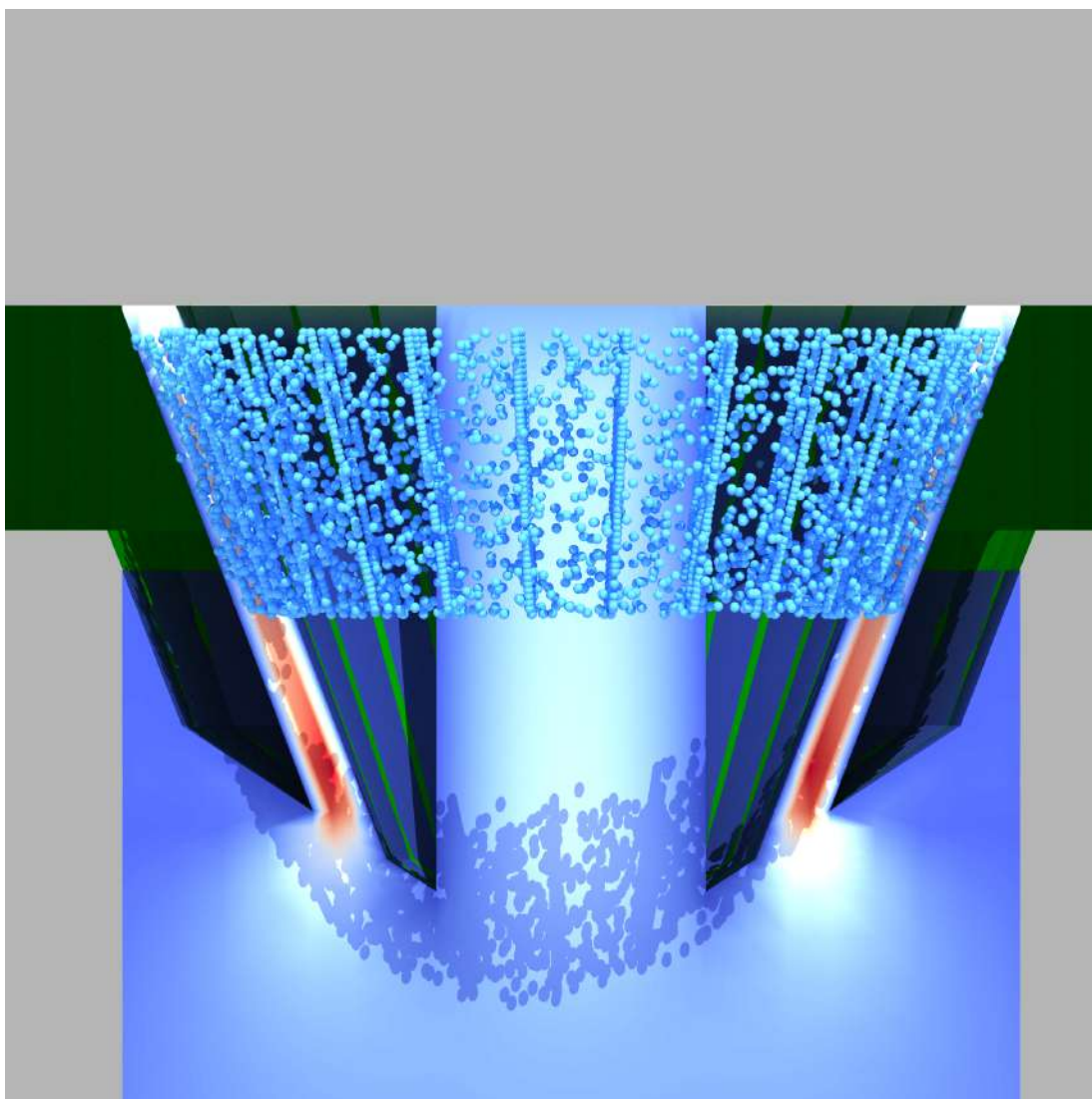


Figura 6.10: Dettaglio punto di entrata particelle con ugello e flusso, sezione laterale.

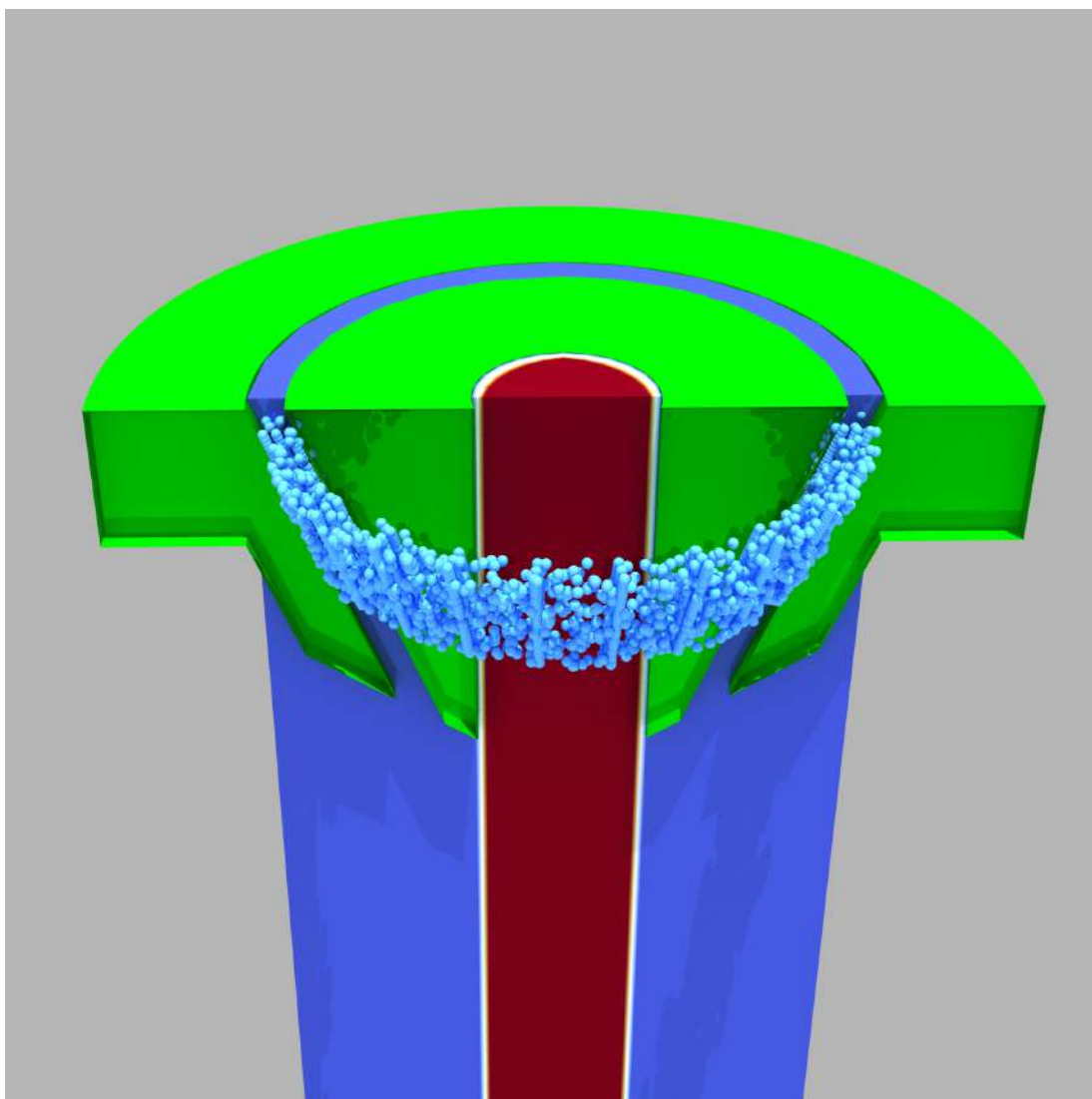


Figura 6.11: Dettaglio punto di entrata particelle con ugello e flusso, sezione laterale.

Conclusioni e sviluppi futuri

Questo progetto di tesi ha visto la nascita di un nuovo tipo di solutore specializzato nella simulazione di processi LMD che utilizza una combinazione di metodi, Lagrangiano e Euleriano, per predire il movimento di un flusso di particelle metalliche. Nonostante i risultati in grado di computare adesso sono numericamente corretti vi sono ancora alcune funzionalità la cui implementazione aumenterebbe certamente il valore del simulatore. Una fra queste sicuramente è l'introduzione di ulteriori algoritmi di parallelizzazioni per garantire tempi di simulazione ancora più bassi.

Ulteriori parallelizzazioni

Al momento viene utilizzato un approccio ibrido per quanto riguarda la distribuzione del carico di lavoro della simulazione su più processi. In particolare nella fase di computazione Euleriana dei flussi viene impiegato un algoritmo di parallelizzazione di calcolo (5.7). Dealii, però, offre diversi strumenti per parallelizzare efficientemente ulteriori carichi di lavoro, tra questi c'è la possibilità di distribuire la mesh, tramite l'oggetto `parallel::distributed::Triangulation< dim, spacedim >` (3.2.1) su più calcolatori per permettere di parallelizzare le operazioni di costruzione o suddivisione nel caso di mesh molto risolte [BBHK11].

Un altro spunto che vale la pena considerare è quello di parallelizzare il metodo Lagrangiano responsabile per la computazione del flusso di particelle metalliche. Questo tipo di processo accoppiato ad alcuni algoritmi accennati nella sezione 4.2.1, offerti dalla libreria Intel TBB, potrebbero ridurre drasticamente i tempi di calcolo. Quest'implementazione certamente richiederebbe uno sforzo implementativo considerevole

in quanto ogni struttura dati andrebbe correttamente gestita per evitare race-condition e thread-lock.

Infine il simulatore gioverebbe sicuramente da un interfaccia grafica capace di gestire correttamente i vari file di input e parametri, così da semplificare un futuro utilizzo che al momento risulta piuttosto ostico a chi non ha partecipato allo sviluppo. Si potrebbe realizzare un semplice wrapper che organizza i file e gestisce l'esecuzione del simulatore in maniera autonoma.

Bibliografia

- [ABB⁺20] Daniel Arndt, Wolfgang Bangerth, Bruno Blais, Thomas C. Clevenger, Marc Fehling, Alexander V. Grayver, Timo Heister, Luca Heltai, Martin Kronbichler, Matthias Maier, Peter Munch, Jean-Paul Pelteret, Reza Rastak, Ignacio Thomas, Bruno Turcksin, Zhuoran Wang, and David Wells. The `deal.II` library, version 9.2. *Journal of Numerical Mathematics*, 2020. in press.
- [ABH⁺15] Martin S. Alnæs, Jan Blechta, Johan Hake, August Johansson, Benjamin Kehlet, Anders Logg, Chris Richardson, Johannes Ring, Marie E. Rognes, and Garth N. Wells. The fenics project version 1.5. *Archive of Numerical Software*, 3(100), 2015.
- [BBHK11] Wolfgang Bangerth, Carsten Burstedde, Timo Heister, and Martin Kronbichler. Algorithms and data structures for massively parallel generic adaptive finite element codes. *ACM Trans. Math. Softw.*, 38:14/1–28, 2011.
- [BDG⁺20] W. Bangerth, J. Dannberg, R. Gassmöller, T. Heister, et al. ASPECT: Advanced Solver for Problems in Earth’s ConvecTion, User Manual. June 2020. doi:10.6084/m9.figshare.4865333.
- [Dus19] Hana Dusíková. 2020 prague meeting invitation and information. 2019.
- [DWZ⁺18] T DebRoy, H.L Wei, J.S Zuback, T Mukherjee, J.W Elmer, J.O Milewski, A.M Beese, A Wilson-Heid, A De, and W Zhang. Additive manufacturing of metallic components – process, structure and properties. *Progress in materials science*, 92(C):112–224, 2018.

- [For94] Message P Forum. Mpi: A message-passing interface standard. Technical report, USA, 1994.
- [Hug00] Thomas J. R Hughes. *The finite element method : linear static and dynamic finite element analysis*. Dover, Mineola (N.Y.), 2000.
- [ISO98] ISO. *ISO/IEC 14882:1998: Programming languages — C++*. September 1998. Available in electronic form for online purchase at <http://webstore.ansi.org/> and <http://www.cssinfo.com/>.
- [ISO17] ISO/IEC. Programming languages — c++. Draft International Standard N4660, March 2017.
- [Kuk07] Alexey Kukanov. The foundations for scalable multicore software in intel threading building blocks. *Intel technology journal*, 11(4), 2007.
- [LORW12] Anders Logg, Kristian B. Ølgaard, Marie E. Rognes, and Garth N. Wells. *FFC: the FEniCS Form Compiler*, chapter 11. Springer, 2012.
- [MM13] Miles Macklin and Matthias Müller. Position based fluids. *ACM Transactions on Graphics (TOG)*, 32(4):104, 2013.
- [MRS19] Jakob M Maljaars, Chris N Richardson, and Nathan Sime. Leopard: a particle library for fenics. *arXiv preprint arXiv:1912.13375*, 2019.
- [TKB16] Bruno Turcksin, Martin Kronbichler, and Wolfgang Bangerth. *Work-Stream* – a design pattern for multicore-enabled finite element computations. *accepted for publication in the ACM Trans. Math. Softw.*, 2016.
- [VRD09] Guido Van Rossum and Fred L. Drake. *Python 3 Reference Manual*. CreateSpace, Scotts Valley, CA, 2009.
- [WJA⁺17] I Wald, Gp Johnson, J Amstutz, C Brownlee, A Knoll, J Jeffers, J Gunther, and P Navratil. Ospray - a cpu ray tracing framework for scientific visualization. *IEEE Transactions on Visualization and Computer Graphics*, 23(1):931–940, 2017.

-
- [WTJF98] H. G. Weller, G. Tabor, H. Jasak, and C. Fureby. A tensorial approach to computational continuum mechanics using object-oriented techniques. *Computers in Physics*, 12(6):620–631, 1998.
- [ZWLS14] Kai Zhang, Shijie Wang, Weijun Liu, and Xiaofeng Shang. Characterization of stainless steel parts by laser metal deposition shaping. *Materials in engineering*, 55:104–119, 2014.