Alma Mater Studiorum
Università di Bologna
Campus di Cesena

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

CONTAINERIZZAZIONE DI APPLICAZIONI HPC

Elaborato in:
High Performance Computing

Relatore:
Prof.
MORENO MARZOLLA

Presentata da:
PIETRO PEZZI

Sessione IV
Anno Accademico 2021-2022

Indice

Sommario

L'incremento della digitalizzazione delle imprese ha portato molte di queste a utilizzare i servizi di Cloud Computing per gestire le loro tecnologie informatiche. Uno dei servizi più utilizzati è quello basato sulla containerizzazione, che consente l'organizzazione e la distribuzione di software in modo automatico.

Con il passare del tempo, le applicazioni informatiche sono diventate sempre più esigenti in termini di potenza di calcolo e tempi di esecuzione veloci, il che ha spinto le aziende interessate a cercare soluzioni al di fuori dei tradizionali servizi di Cloud Computing.

Una possibile soluzione potrebbe essere quella di combinare il Cloud Computing con l'High Performance Computing (HPC), una branca dell'informatica dedicata allo sviluppo di applicazioni in grado di raggiungere i massimi livelli di prestazioni. L'HPC fa leva sulle più recenti tecnologie hardware e software, e utilizza tecniche di parallelizzazione per migliorare le prestazioni.

Tuttavia, data la differenza tra queste due tecnologie, trovare un punto di incontro è stato difficile.

In questo elaborato è stata verificare la validità di Apptainer come soluzione al problema, valutando la sua capacità di fungere da sistema di containerizzazione per le applicazioni HPC. Analizzando e descrivendo lo sviluppo di una applicazione poi containerizzata con tale sistema di containerizzazione.

Capitolo 1

Introduzione

La costante evoluzione delle tecnologie informatiche sta portando alla crescente digitalizzazione delle imprese. Questo processo consiste nell'utilizzo di tecnologie digitali per automatizzare alcune aree aziendali con l'obiettivo di migliorare l'efficienza e ridurre i costi. Tuttavia, la digitalizzazione comporta anche investimenti significativi, ma i vantaggi che offre ne giustificano la spesa.

Ognuno di questi interventi richiederà una quantità adeguata di potenza di calcolo in base alla complessità del servizio. Nella maggior parte dei casi è l'impresa stessa che si occupa dell'installazione e manutenzione del sistema: questo comprende l'acquisto delle componenti hardware richieste e l'individuazione del personale dedicato alla gestione e manutenzione dell'intervento. Tuttavia, con l'aumentare della complessità del servizio, questo approccio potrebbe non essere conveniente. La soluzione più diffusa a questo problema è affidare i compiti di installazione e manutenzione della componente computazionale ad un servizio di *Cloud computing*.

Il termine **Cloud Computing** si riferisce sia alle applicazioni fornite come servizi via Internet, sia all'hardware e al software di sistema presenti nei data center che forniscono tali servizi.

Per rendere la definizione più chiara, si può affermare che il Cloud Computing è un modello di erogazione di servizi informatici che prevede l'utilizzo di risorse presenti su server remoti, accessibili via internet. Ciò consente agli utenti di accedere a software e dati in remoto, senza doverli installare o gestire localmente. In particolare, il termine Cloud si riferisce all'insieme di componenti hardware e software che costituiscono il data center su cui sono eseguiti i vari servizi. Il Cloud è dunque un'infrastruttura complessa che include: server, storage, networking, sistemi di sicurezza e gestione, nonché il software necessario per garantire l'erogazione dei servizi. Tale architettura consente di realizzare un ambiente altamente scalabile e affidabile, in grado di gestire grandi volumi di dati e di traffico Internet.

Quando un'infrastruttura Cloud viene resa disponibile al pubblico a pagamento, viene definita come *Public Cloud*. Inoltre, l'insieme dei servizi venduti tramite il Public Cloud prende il nome di *Utility Computing*. In sostanza, attraverso l'Utility Computing, gli utenti possono utilizzare le risorse informatiche in modo flessibile e pagare solo per ciò che effettivamente consumano.

I servizi di Utility Computing resi disponibli da un sistema di Cloud Computing sono diversi, i due principali tipi di servizi più utilizzati sono la disponibilità di grande spazio di archiviazione per dati e la vendita di potenza di calcolo tramite l'acquisto di server virtuali.

In questo elaborato, ci concentreremo principalmente sull'insieme di servizi di Utility Computing che vendono potenza di calcolo, ovvero quei servizi che le aziende utilizzano per accedere a diverse unità di calcolo o server virtuali all'interno del Cloud, al fine di eseguire applicazioni software.

Per meglio comprendere i concetti usati nel seguito di questo elaborato, analizziamo le principali tecnologie adottate da questi servizi di Cloud Computing per la gestione e vendita di potenza di calcolo.

1.1 Virtualizzazione

Un affidabile servizio di Cloud Computing dispone di una quantità limitata di risorse di calcolo. Per ottimizzare la gestione e la distribuzione di queste risorse, viene utilizzata la tecnologia di virtualizzazione. Questa tecnologia consiste nella creazione di un ambiente virtuale che simula un sistema operativo, permettendo l'utilizzo di più calcolatori virtuali su un unico dispositivo fisico. La gestione di questi calcolatori virtuali è effettuata dalla componente *Hypervisor* del sistema di virtualizzazione. Esistono diverse soluzioni software di virtualizzazione: tra le più comuni abbiamo *VMWare* e *VirtualBox*, ma è possibile che un'azienda di Cloud Computing utilizzi un software proprietario sviluppato internamente.

La virtualizzazione può essere di due tipi principali, a seconda di dove viene eseguita la componente *Hypervisor*. Nella virtualizzazione di tipo uno, l'*Hypervisor* viene eseguito direttamente sul sistema hardware e pertanto viene chiamata anche virtualizzazione bare-metal. Nel secondo tipo, invece, l'*Hypervisor* viene

eseguito su un sistema operativo ospitante, e questo spiega il nome *Hosted Virtualization*. Entrambi i tipi di virtualizzazione sono raffigurati in Figura 1.1.

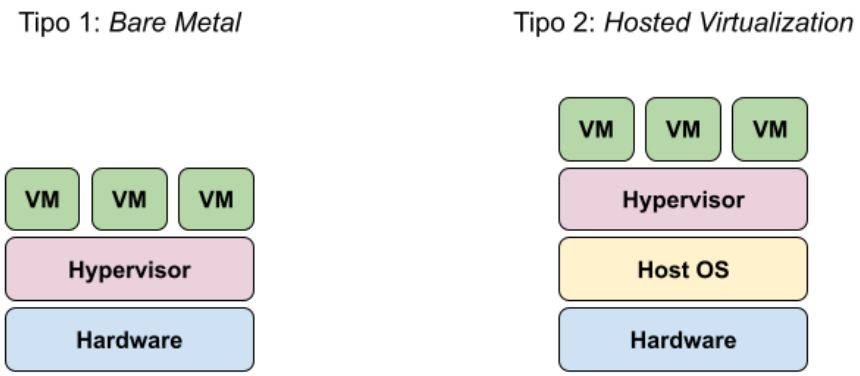

Figura 1.1: Rappresentazione dei due tipi di virtualizzazione

Questi sistemi di virtualizzazione rendono possibile l'installazione e la distribuzione automatica di macchine virtuali, rendendo così la vendita di potenza di calcolo più semplice ed efficiente. Adottando questo metodo, una azienda può anche garantire un utilizzo più sicuro dei suoi calcolatori. La gestione delle macchine virtuali da parte dell'*Hypervisor* permette agli amministratori di gestire i permessi degli utenti, prevenendo così eventuali danni all'hardware sottostante o al sistema operativo ospitante.

1.2 Containerizzazione

La Containerizzazione consiste nel raggruppare il codice di una specifica applicazione insieme alle sue dipendenze e librerie richieste per il suo funzionamento, in modo da renderlo essenzialmente isolato all'interno di un container. Questo processo consente di eseguire il container su qualsiasi sistema che supporti

lo stesso sistema di containerizzazione utilizzato per la containerizzazione dell'applicazione. Il sistema di containerizzazione più utilizzato al momento è *Docker*. La proprietà "*containerizza una volta ed esegui ovunque*" viene adottata dai servizi di Cloud Computing, fornendo una soluzione conveniente per eseguire container su qualsiasi macchina con il giusto sistema di containerizzazione.

Similmente alla virtualizzazione, la containerizzazione è gestita da una componente chiamata *Container Engine*, responsabile della gestione dei vari container in esecuzione. Per l'esecuzione di tale Container Engine è necessario un sistema operativo ospitante, che nella maggior parte dei casi è un sistema operativo Linux. Per sfruttare al meglio le risorse disponibili i servizi di Cloud computing adottano soluzioni ibride, utilizzando la virtualizzazione per la distribuzione di macchine virtuali, e all'interno di queste concedono l'esecuzione di applicazioni containerizzate, come descritto in figura 1.2.

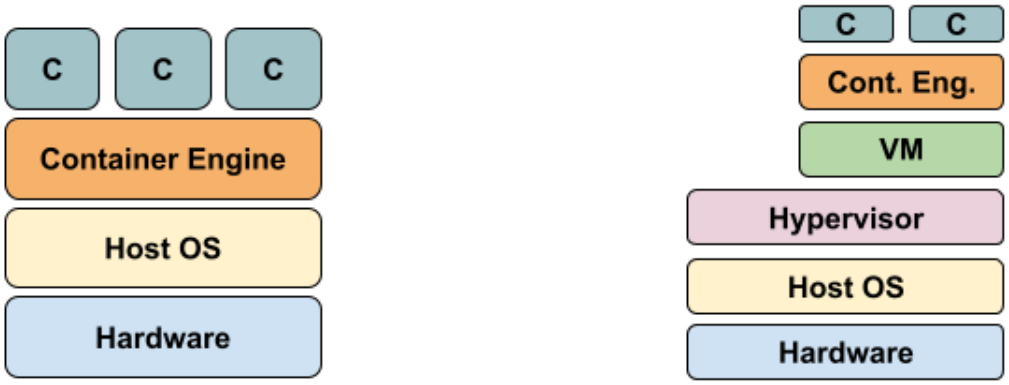

Figura 1.2: Raffigurazione containerizzazione a sinistra, soluzione ibrida di Virtualizzazione e containerizzazione a destra.

Le principali differenze tra l'utilizzo di un Container Engine e un Hypervisor sono la sicurezza e le prestazioni. L'utilizzo di un Hypervisor comporta la virtualizzazione di un intero sistema operativo, il che può risultare in una diminuzione delle prestazioni, poiché la virtualizzazione del sistema operativo rallenta le prestazioni delle applicazioni eseguite all'interno della macchina virtuale. Tuttavia, l'utilizzo dell'Hypervisor risulta essere molto sicuro, in quanto rende il sistema ospitante al sicuro da eventuali danni che potrebbero essere causati dall'interno della macchina virtuale. D'altra parte, i sistemi di containerizzazione virtualizzano solo l'ambiente strettamente necessario all'esecuzione dell'applicazione, garantendo tempi di avvio e stop molto rapidi rispetto alla virtualizzazione. Tuttavia, i sistemi di containerizzazione possono presentare problemi di sicurezza che rendono il loro utilizzo più complesso rispetto a quello dell'Hypervisor. In sintesi, l'Hypervisor è maggiormente adatto per la creazione di ambienti completamente isolati e sicuri, mentre i container sono più adatti per l'esecuzione di applicazioni in modo più efficiente e leggero, ma con un livello di sicurezza inferiore rispetto all'Hypervisor.

I servizi di Cloud Computing, adottando le tecnologie precedentemente descritte, vengono in soccorso alle piccole e medie imprese prendendosi in carico la gestione delle risorse hardware e software da esse richieste. Tuttavia, con l'aumento della complessità delle applicazioni, i requisiti sono diventati più impegnativi.

Negli ultimi anni questa difficoltà è ulteriormente aumentata con l'evoluzione dell'Intelligenza Artificiale e applicazioni di Machine Learning. In sintesi, le applicazioni di Intelligenza Arti-

ficiale (AI) sono sistemi software progettati per eseguire compiti che normalmente richiederebbero l'intervento umano: come la classificazione di dati, la traduzione di lingue, la comprensione del linguaggio naturale, e molto altro. Queste applicazioni sono alimentate da algoritmi di apprendimento automatico e utilizzano grandi quantità di dati per migliorare continuamente la loro precisione e capacità di prendere decisioni.

L'elaborazione di grandi quantità di dati, noti come parametri, richiede una notevole quantità di potenza di calcolo. I servizi di Cloud Computing hanno implementato soluzioni specializzate per soddisfare questa domanda, ma la necessità di rapide decisioni da parte delle applicazioni AI richiede anche un'ottimizzazione del loro tempo di esecuzione. Questa sfida ha portato alla ricerca di soluzioni al di fuori del Cloud Computing tradizionale e quindi all'interesse per l'High Performance Computing.

1.3 High Performance Computing

L'High Performance Computing (HPC) è una disciplina tecnologica che si concentra sull'utilizzo di sistemi informatici altamente sofisticati per risolvere problemi complessi in modo efficiente. Questi sistemi combinano hardware di alta qualità, software avanzato e tecniche di calcolo parallelo per ottenere prestazioni ottimali.

Ci interessa particolarmente la programmazione parallela, ovvero un modello di programmazione che mira a migliorare le prestazioni di una applicazione tramite l'esecuzione parallela di codice. In questa relazione, verrà analizzata l'esecuzione parallela sui processori tramite l'utilizzo della libreria MPI e la pro-

grammazione parallela su GPU NVIDIA mediante la tecnologia CUDA.

Le strutture di calcolo HPC, meglio conosciute come HPC Cluster, sono impianti progettati appositamente per il raggiungimento dell'obiettivo HPC. Rispetto alle strutture di Cloud Computing, questi HPC cluster presentano una struttura molto diversa. Le strutture HPC sono progettate per ridurre al minimo la latenza delle comunicazioni tra processi o thread durante l'esecuzione di programmi in parallelo. Questo viene fatto attraverso l'uso di efficienti meccanismi di comunicazione sviluppati appositamente. A differenza delle strutture Cloud, dove le comunicazioni tra i vari componenti durante l'esecuzione dell'applicativo sono considerate meno rilevanti, se non addirittura assenti. Inoltre, a causa della complessa natura di questi HPC Cluster, non è possibile installare facilmente le dipendenze necessarie per l'esecuzione di comuni applicazioni. Questo potrebbe causare problemi di compatibilità con altre librerie preinstallate e rendere il sistema vulnerabile a nuove minacce alla sicurezza.

La soluzione al problema descritto in precedenza si trova in un equilibrio tra le tecnologie utilizzate dal Cloud Computing e l'HPC. Da un lato, si vuole mantenere la convenienza della containerizzazione, ovvero la semplicità nello sviluppo e nella distribuzione delle applicazioni; dall'altro lato, si desidera mantenere le caratteristiche dell'HPC, in particolare le buone prestazioni ottenibili attraverso il calcolo parallelo e l'esecuzione di applicazioni sugli HPC Clusters. Recentemente, questo equilibrio è stato reso possibile con l'utilizzo del sistema di containerizzazione Apptainer [6].

1.4 Apptainer

Apptainer, originariamente conosciuto come Singularity, è un sistema di containerizzazione appositamente progettato per le applicazioni HPC. Si distingue per la capacità di eseguire applicazioni containerizzate con prestazioni elevate paragonabili a quelle bare metal. Inoltre, per consentirne l'utilizzo anche su HPC Cluster, Apptainer garantisce un elevato livello di sicurezza, permettendone un utilizzo sicuro anche da parte di utenti che non dispongono di permessi amministrativi.

Attualmente, Apptainer è riconosciuto come il sistema di containerizzazione per applicazioni ad alte prestazioni (HPC) più diffuso. Viene attualmente impiegato in una vasta gamma di cluster HPC in tutto il mondo (Il sito ufficiale di apptainer mostra una lista di alcuni di questi HPC Centers nella sezione "*Quick start*" della documentazione [1]).

La parallelizzazione di un'applicazione è un compito complesso, quindi potrebbe sembrare che anche la sua containerizzazione sia difficile. Tuttavia, il sistema di containerizzazione Apptainer è stato progettato appositamente per semplificare questa attività, richiedendo solo una conoscenza di base del funzionamento della containerizzazione. Inoltre, la containerizzazione con Apptainer è ancora più semplice per le applicazioni MPI e per quelle eseguibili su schede video NVIDIA, grazie al pieno supporto di queste tecnologie da parte di Apptainer.

Per verificare la validità di questo sistema di containerizzazione come possibile soluzione al problema della cresente domanda per potenza di calcolo e bassi tempi di esecuzione, questo elaborato analizza e discute il processo di containerizzazione di appli-

cazioni High Performance utilizzando Apptainer. Descrivendo lo sviluppo e parallelizzazione di un applicativo poi containerizzato con tale sistema di containerizzazione. Concludendo studiando i cambiamenti nelle prestazioni causati dalla containerizzazione e quindi analizzando i dati raccolti.

Capitolo 2

Sviluppo applicazione e implementazioni parallele

Come applicazione per la verifica del sistema di containerizzazine Apptainer è stata scelta l'implementazione di un algoritmo di filtraggio digitale, ovvero il Filtro Bilaterale [5]. La scelta di questo algoritmo è dovuta alla sua appartenenza al settore dell'*Image denoising*, un campo fondamentale nell'elaborazione di immagini e visione artificiale. Poiché questi due ambiti richiedono spesso una elevata potenza di calcolo a causa della complessità degli algoritmi studiati, l'implementazione del Filtro Bilaterale è coerente con il problema discusso nel capitolo Introduzione.

Il codice di tutte le implementazioni descritte in seguito è disponibile presso un repository pubblico su Github [2].

2.1 Filtro bilaterale

Un algoritmo di filtraggio digitale per le immagini applica una determinata sequenza di operazioni matematiche a ogni pixel dell'immagine, ottenendo così una nuova versione dell'immagine

con diverse qualità. Nel caso dei filtri digitali per la rimozione del rumore come il Filtro bilaterale, lo scopo di queste operazioni è quello di rimuovere alcune imperfezioni causate da disturbi.

Il Filtro Bilaterale rimuove questo rumore aggiornando il valore di ogni pixel con una media pesata del suo *vicinato*. Ponendo $I_O(x)$ il valore originale del pixel x e $I(x)$ il suo valore aggiornato, il filtro adotta la seguente formula:

$$I(x) = \frac{1}{w} \sum_{x_i \in \Omega} I_O(x_i) \cdot f_r(\|I_O(x_i) - I_O(x)\|) \cdot g_s(\|x_i - x\|) \quad (2.1)$$

con

$$w := \sum_{x_i \in \Omega} f_r(\|I_O(x_i) - I_O(x)\|) \cdot g_s(\|x_i - x\|)$$

Dove l'insieme $x_i \in \Omega$ comprende tutti i pixel all'interno del *vicinato* del pixel x (Rappresentazione del vicinato di un pixel in figura 2.1). Infine, f_r e g_s sono le funzioni che determinano la somiglianza tra le intensità di colore dei due pixel e il peso dato dalla distanza tra di essi.

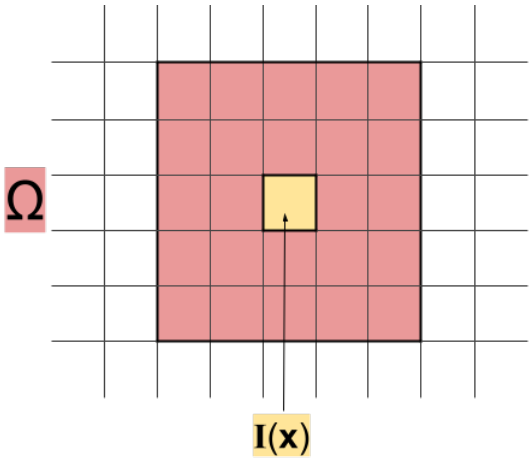

Figura 2.1: Rappresentazione del vicinato (Ω) di un pixel x.

Data la quantità di operazioni necessarie per calcolare il valore di correzione di ogni pixel, il filtro bilaterale risulta essere lento, soprattutto con immagini ad alta risoluzione. Sebbene esistano diverse implementazioni migliorate dell'algoritmo, dato lo scopo del progetto, il tempo di esecuzione del filtro bilaterale è stato migliorato tramite tecniche di Programmazione Parallela.

2.2 Implementazione seriale

Il linguaggio di programmazione scelto per lo sviluppo del progetto è C, principalmente dovuto alle librerie di programmazione parallela che si intendono utilizzare. Lo sviluppo dell'implentazione seriale ha compreso anche la scittura di un semplice programma dedicato alla lettura e scrittura di immagini, utilizzando il formato PPM (Portable Pixel Map), data la sua facilità di lettura dei file. Inoltre, per verificare la validità delle varie implementazioni e simularne un loro utilizzo pratico, è stato creato un ulteriore programma per l'aggiunta di rumore alle immagini. Il rumore in questione è un disturbo molto comune nelle immagini, ovvero il rumore Gaussiano; causato da una scarsa illuminazione o alte temperature durante l'acquisizione di un'immagine (Un esempio di questo rumore è presente in figura 2.2)

Per la generazione di tale rumore il programma utilizza la Trasformazione di Box-Muller, ovvero un metodo per la generazione di numeri casuali distribuiti gaussianamente con media nulla e varianza uno. In particolare, utilizza una forma derivata dalla forma base del metodo:

$$X = (\sqrt{-2 \ln s_1} \cdot \cos(2\pi s_2) \cdot \sigma) + \mu \tag{2.2}$$

La formula 2.2 utilizza i due campioni s_1 e s_2, ovvero numeri generati casualmente all'interno dell'intervallo $(0, 1)$. Questa formula, rispetto alla forma base, permette di regolare il rumore applicato tramite i parametri media (μ) e deviazione standard (σ). Questa quantità X è poi casualmente aggiunta o sottratta al valore di ogni pixel, simulando così il rumore Gaussiano.

Figura 2.2: Applicazione rumore gaussiano, con $\mu = 10$ e $\sigma^2 = 30$.

In relazione all'implementazione del filtro bilaterale, il programma è stato sviluppato con conformità alla definizione dell'algoritmo descritto nella sezione 2.1. In particolare, vi sono alcuni dettagli implementativi riguardanti le funzioni f_s e g_s usate nell'equazione 2.1. La distanza tra due pixel è calcolata utilizzando la formula della distanza euclidea tra due punti. Per quanto riguarda la determinazione della somiglianza dell'intensità di colore tra due pixel e il peso della loro distanza, il programma utilizza in entrambi i casi la funzione kernel gaussiana:

$$G(x, \sigma) = \frac{1}{\sqrt{2\pi}\sigma} e^{-\frac{x^2}{2\sigma^2}} \tag{2.3}$$

In conclusione, l'implementazione seriale del filtro bilaterale è basata su tre parametri fondamentali:

- *radius*: Definisce la distanza in pixel dal centro della finestra al bordo, e viene utilizzato per calcolare la dimensione del *vicinato* di un pixel (Nella figura 2.1 è rappresentato un vicinato di radius 2).
- σ_c (*sigma color*): Regola il range di somiglianza dell'intensità di colore tra due pixel, ed è utilizzato nella funzione 2.3 per questo scopo.
- σ_s (*sigma spatial*): Regola il peso dato dalla distanza tra due pixel, e viene anch'esso utilizzato nella funzione 2.3 per questo scopo.

La seguente figura 2.3 mostra un risultato dell'applicazione del filtro bilaterale.

Figura 2.3: Immagine originale, con rumore gaussiano e dopo applicazione filtro bilaterale con $radius = 10$, $\sigma_c, \sigma_s = 30$.

2.3 Implementazione MPI

Per migliorare le prestazioni dell'implementazione seriale del filtro, la prima tecnica di programmazione parallela adottata consiste nell'esecuzione parallela del codice su più processori utilizzando la libreria MPI (Message Passing Interface).

MPI è una libreria che gestisce l'esecuzione parallela di codice su architetture a memoria distribuita. Una architettura a memoria distribuita consiste in più unità di calcolo in grado di eseguire istruzioni autonomamente, ognuna con la propria memoria dedicata (Rappresentazione architettura in figura 2.4).

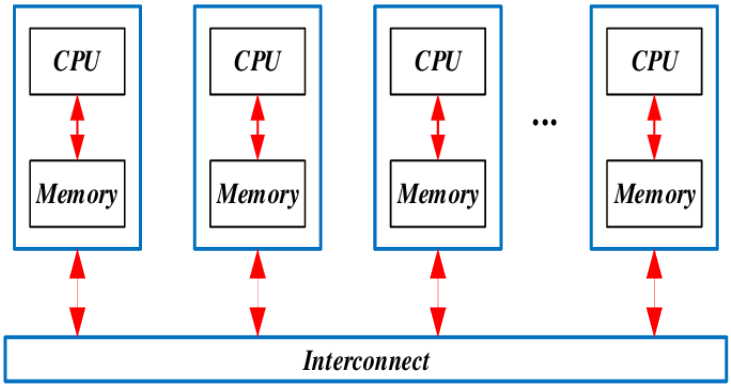

Figura 2.4: Rappresentazione architettura a memoria distribuita. Fonte [3]

Le unità di calcolo comunicano tra di loro tramite un adeguato canale di interconnesione. La libreria MPI fornisce diverse funzioni per gestire la comunicazione e lo scambio di dati tra questi processi.

Per la realizzazione di questa implementazione, sono state utilizzate le seguenti funzioni:

- `MPI_Scatterv`: Questa funzione consente di distribuire in modo efficiente un array di dati da un processo radice a un insieme di processi destinatari. Generalmente, il compito di agire come processo radice per questa operazione è affidato allo stesso processo che si occupa delle altre operazioni del programma che non richiedono esecuzione parallela.

- `MPI_Sendrecv`: Questa funzione consente lo scambio di una quantità di dati specifica tra due processi. Entrambi i processi coinvolti nello scambio invieranno e riceveranno una quantità di dati definita.

- `MPI_Gatherv`: La funzione `MPI_Gatherv` funge come il complemento inverso della funzione `MPI_Scatterv`. In questa funzione, un processo radice è responsabile di raccogliere in modo efficiente un insieme di dati provenienti da più processi sorgenti e memorizzarli in un singolo array nel processo destinatario.

L'implementazione MPI, partendo dal codice dell'implementazione seriale descritta nella sezione 2.2, migliora le prestazioni del programma distribuendo equamente il carico di lavoro ai processi. Considerando un'immagine in ingresso con altezza M e larghezza N, e indicando con p il numero di processi, si può definire il massimo numero di pixel assegnabili a ciascun processo come segue:

$$n_{max} := \lceil \frac{M \cdot N}{p} \rceil$$

Utilizzando questa unità per la distribuzione dei pixel, i primi $p - 1$ processi si occuperanno della computazione di n_{max} pixel, lasciando i pixel residui al processo p.

La distribuzione di questi pixel viene eseguita attraverso una semplice operazione uno a molti, sfruttando la funzione `MPI_Scatterv` (La Figura 2.5 mostra un esempio di distribuzione dei pixel di una immagine).

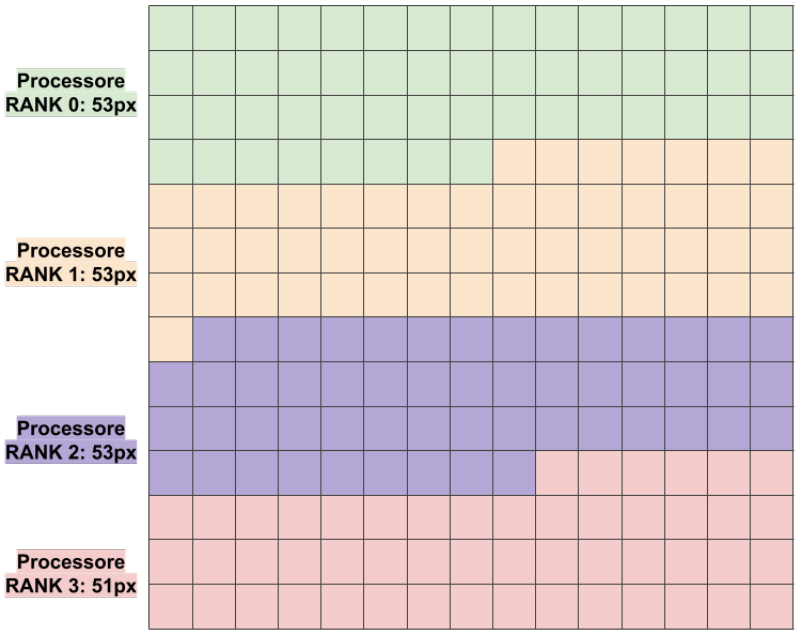

Figura 2.5: Distribuzione dei pixel tra 4 processori con una immagine 15x14.

Dato che l'algoritmo del Filtro Bilaterale esegue per ogni pixel operazioni su un vicinato di raggio *radius*, ogni processo necessiterà dei pixel contenuti nel vicinato che non si trovano all'interno della propria porzione.

Ogni processo allocherà una quantità di memoria sufficiente per il contenimento dei pixel computabili e dei *ghost pixel*, ovvero i pixel richiesti non presenti nella propria porzione.

A partire dall'indice I_{start}, ovvero l'inizio della porzione di un processo all'interno dell'immagine, viene calcolata la quantità di *ghost pixel* precedenti a tale porzione.

Si individua la distanza tra I_{start} e l'inizio della riga con:

$$d_{start} := I_{start} - \lfloor \frac{I_{start}}{N} \rfloor \cdot N$$

Ottenuta tale quantità si calcolano i *ghost pixel* precedenti eseguendo la somma:

$$ghost_{prev} := d_{start} + (radius \cdot N)$$

Nella stessa maniera vengono determinati i *ghost pixel* successivi; dato l'indice I_{end} si calcola la distanza tra tale indice e fine riga con:

$$d_{post} := (((\lfloor \frac{I_{end}}{N} \rfloor + 1) \cdot N) - 1) - I_{end}$$

Esattamente come per i *ghost pixel* precedenti, I *ghost pixel* successivi si ottengono con la somma:

$$ghost_{post} := d_{post} + (radius \cdot N)$$

Considerando un processo con n_{max} pixel assegnati, tale processo occuperà $ghost_{prev} + n_{max} + ghost_{post}$ pixel di memoria.

La Figura 2.6 riassume i concetti precedentemente descritti.

La ragione per cui vengono memorizzati anche i pixel d_{prev} e d_{post} è dovuta alla funzione per l'applicazione del Filtro Bilaterale. Tale metodo esegue l'indicizzazione sulla porzione fornita sfruttando la funzione `IDX`. Dato un indice di riga e uno di colonna, la funzione `IDX` restituisce il corrispondente indice lineare su una griglia M x N. Le due quantità servono quindi per normalizzare la dimensione della matrice corrispondente alla porzione di ogni processo in una matrice rettangolare $\tilde{M}$ x N, con:

$$\tilde{M} := \frac{ghost_{prev} + n_{max} + ghost_{post}}{N}$$

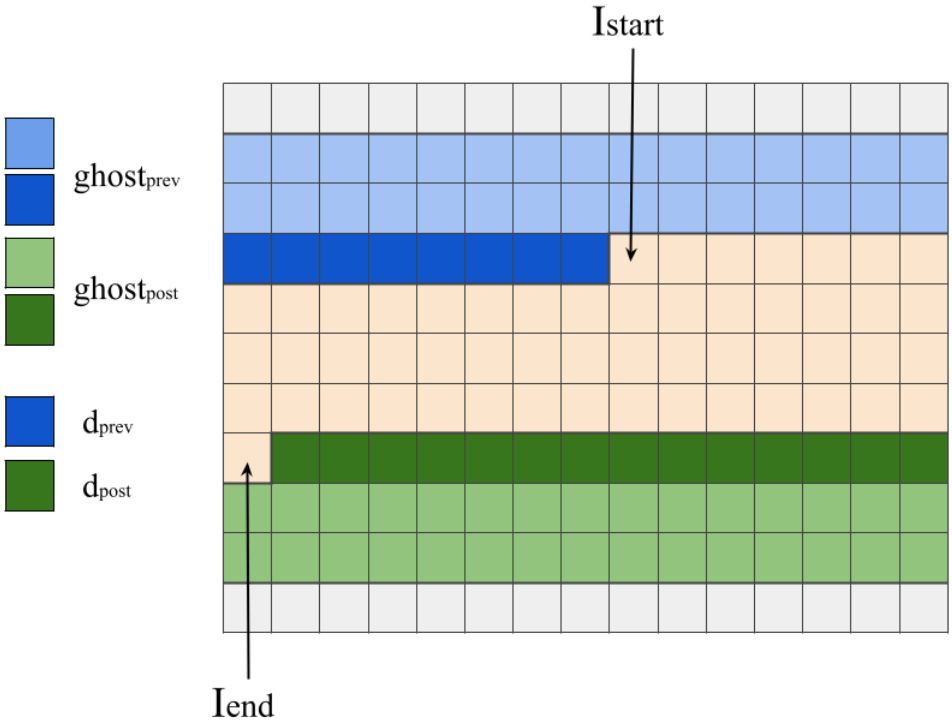

Figura 2.6: Esempi *ghost pixel* precedenti e successivi necessari a una porzione, con $radius$ =2.

Come descritto precedentemente, ogni processo possiede un massimo di n_{max} pixel computabili. Affinché ogni processo possieda i *ghost pixel* da inviare ai processi vicini, il programma effettua un controllo sul valore di radius. Tale quantità dovrà rispettare la condizione:

$$radius < \lfloor \frac{M}{p} \rfloor - 1$$

con M pari all'altezza dell'immagine e p al numero di processi.

Il trasferimento dei *ghost pixel* successivi e precedenti avviene tramite due operazioni punto a punto, entrambe implementate

utilizzando la funzione `MPI_Sendrecv`. Nel primo trasferimento ogni processo invia al vicino successivo i pixel precedenti da esso richiesti, mentre nel secondo trasferimento ogni processo invia al vicino precedente i *ghost pixel* successivi.

Una volta ottenuti tutti i pixel necessari, ogni processo applica l'algoritmo del Filtro Bilaterale sulla propria porzione. Completata l'operazione, le porzioni aggiornate sono inviate al processo 0 utilizzando la funzione `MPI_Gatherv`. Tale processo si occuperà di ricostruire l'immagine filtrata e della sua scrittura.

2.4 Implementazione CUDA

Per ottenere prestazioni migliori nelle applicazioni di elaborazione di immagini ad alta risoluzione, è consigliabile eseguire l'applicazione in parallelo su schede video. Per comprendere meglio questa implementazione, è necessario analizzare le architetture delle Schede video e il funzionamento della libreria CUDA utilizzata.

Le **GPU** (Graphics Processing Units) sono una soluzione hardware ideale per l'esecuzione di algoritmi che richiedono molte operazioni, grazie alla grande quantità di unità di calcolo disponibili. Anche se le unità di calcolo nelle GPU possiedano una potenza computazionale inferiore rispetto a quelle nelle CPU, la loro notevole quantità le rende particolarmente adatte per l'esecuzione parallela di algoritmi che richiedono operazioni ripetitive e una coordinazione limitata o nulla tra le istruzioni.

Per illustrare la differenza di architettura tra CPU e GPU, si può fare riferimento alla figura 2.7.

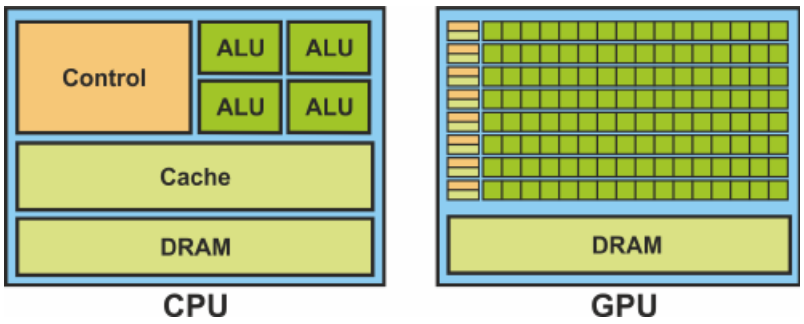

Figura 2.7: Confronto tra l'architettura di una CPU e quella di una GPU. Fonte [4]

L'algoritmo del Filtro Bilaterale, così come la maggior parte degli algoritmi di filtraggio digitale, possiede tutte le caratteristiche ideali per l'esecuzione parallela su GPU.

In questo progetto è stata utilizzata la piattaforma CUDA, un sistema che rende possibile l'esecuzione di codice parallelo su schede video NVIDIA. Nella piattaforma CUDA, l'unità minima di lavoro eseguibile sulla scheda video è chiamata *thread*. I thread sono organizzati in blocchi, che a loro volta sono raggruppati in griglie (Rappresentazione di questi raggruppamenti in figura 2.8).
Sebbene i thread costituiscano l'unità minima di lavoro, i blocchi costituiscono le unità indipendenti di lavoro.

In questa implementazione, è importante anche descrivere le due partizioni di memoria della GPU utilizzate dal programma sviluppato:

- **Memoria globale**: è la memoria più grande e accessibile a tutti i blocchi e quindi a tutti i thread disponibili. Grazie alla sua accessibilità, la memoria globale può contenere fino a diverse decine di gigabyte di dati.

- **Memoria condivisa**: è una memoria accessibile solo ai thread all'interno di un blocco. Poiché questa memoria è condivisa solo tra i thread all'interno del blocco, è molto più piccola della memoria globale. Tuttavia, presenta il vantaggio di un tempo di lettura e scrittura dei dati molto più veloce rispetto alla memoria globale.

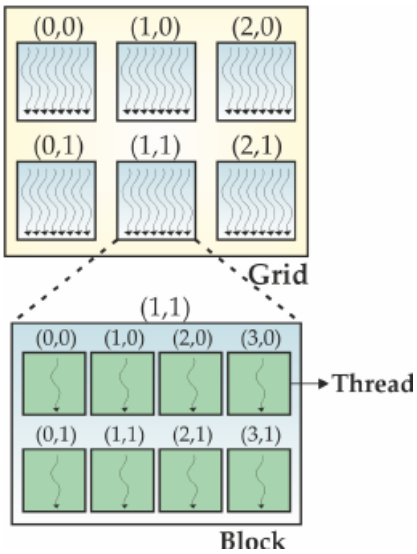

Figura 2.8: Elaborazione sulla GPU: griglie composte da blocchi con thread di calcolo.
Fonte [4]

Un altro dettaglio della piattaforma CUDA è la distinzione presente tra il codice eseguito dalla CPU (*host*) e quello eseguito sulla scheda video (*device*). Non tutte le istruzioni di un'applicazione richiedono di essere eseguite in parallelo; pertanto, queste istruzioni sono eseguite dalla CPU. Al contrario, le istruzioni eseguite sulla scheda video sono contenute in funzioni apposite chiamate *kernel functions*. Queste funzioni sono delimitate dalla parola chiave `__device__`.

Il programma inizia caricando l'immagine da elaborare nella memoria globale della scheda video. Successivamente, viene riservato uno spazio di memoria globale sulla GPU dove sarà memorizzato l'immagine filtrata.

L'implementazione CUDA migliora le prestazioni del programma elaborando l'immagine in blocchi, ognuno dei quali è costituito da `BLKDIM` x `BLKDIM` pixel. Data la quantità di variabili necessarie per l'applicazione del filtro bilaterale a un singolo pixel, il programma utilizza `BLKDIM` = 16, ovvero ogni blocco sarà costituito da 256 thread.

Ogni thread si occupa dell'elaborazione di un singolo pixel; per eseguire tale operazione è necessario utilizzare i vari pixel contenuti nel vicinato di raggio `RADIUS` attorno al pixel in esame. Al fine di ridurre il numero di letture dalla memoria globale necessarie per l'ottenimento di questi *ghost pixel*, il programma sfrutta la memoria condivisa di ogni blocco.

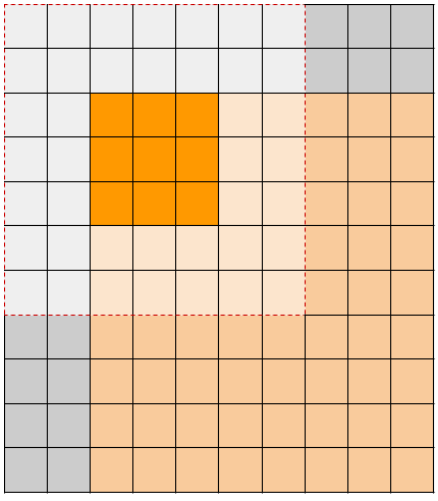

Figura 2.9: Rappresentazione dei rispettivi pixel caricati da ogni thread in memoria locale.

Prima dell'applicazione del filtro, ogni pixel si occupa di caricare una determinata porzione dell'immagine in memoria globale in memoria locale: tale porzione dipende dall'indice locale di ogni thread.

La prima operazione, effettuata da ogni thread indipendentemente dal loro indice, consiste nel caricamento del pixel su cui applicare il filtro (Primo caricamento rappresentato in figura 2.9).

I thread corrispondenti ai pixel situati sui bordi della porzione si occuperanno di caricare anche i `RADIUS` ghost pixel vicini (Secondo caricamento rappresentato in figura 2.10).

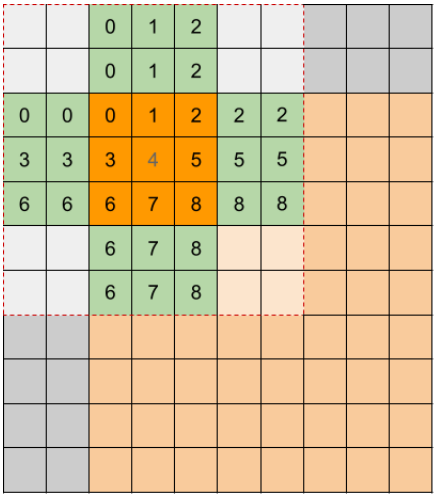

Figura 2.10: Caricamento dei *ghost pixel* in prossimita dei bordi.

Infine, i rimanenti pixel necessari vengono caricati dai thread corrispondenti ai pixel situati sugli spigoli della porzione (Terzo caricamento rappresentato in figura 2.11).

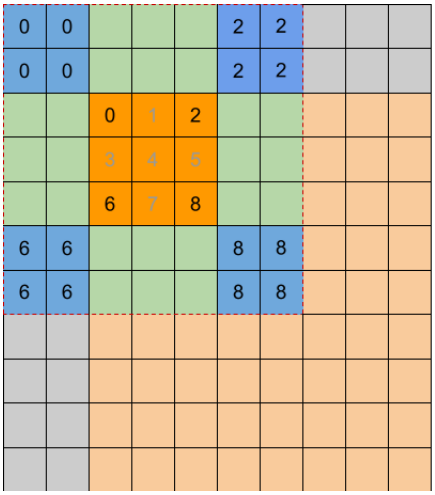

Figura 2.11: Caricamento dei *ghost pixel* in prossimita degli *spigoli*.

Una volta caricati tutti i pixel in memoria condivisa, i thread di ogni blocco vengono sincronizzati e quindi viene applicato il filtro bilaterale. Il nuovo valore di ogni pixel viene quindi salvato nel nuovo array precedentemente allocato in memoria globale.

Al termine del programma, l'immagine filtrata viene letta dalla memoria globale della scheda video e salvata in formato file.

La dimensione di `BLKDIM` precedentemente definita comporta alcune restrizioni sulle immagini trattabili dal programma. In particolare l'implementazione richiede che altezza e larghezza delle immagini siano uguali e multipli di `BLKDIM`.

Capitolo 3

Processo di containerizzazione delle implementazioni parallele

Il sistema di containerizzazione Apptainer supporta la containerizzazione di applicazioni MPI e applicazioni eseguibili su schede video (CUDA). Per comprendere come le due implementazioni siano state containerizzate, è necessario conoscere alcuni concetti di base della containerizzazione.

Il funzionamento di un qualsiasi sistema di containerizzazione è basato su un elemento fondamentale: le *Immagini di Container*. Queste immagini includono tutti gli elementi necessari per far funzionare un'applicazione, ovvero: il codice sorgente, le dipendenze, le librerie e le configurazioni. Utilizzare un'immagine di container garantisce che l'applicazione funzionerà in maniera uniforme in qualsiasi ambiente con il sistema di containerizzazione intallato, poiché tutti gli elementi necessari sono racchiusi all'interno del contenitore stesso.

Le immagini di container possono essere ottenute in due modi: scaricandole da piattaforme di hosting dedicate ai container, come *DockerHub*, oppure tramite *File di Definizione*. Questi fi-

le permettono di costruire immagini di container personalizzate tramite una serie di istruzioni appropriate.

In Apptainer, le immagini di container prendono il nome di: *Singularity Image File* (`.sif`). Per la containerizzazione di entrambe le implentazioni, le immagini di container utilizzate sono state ottunete sviluppando un opportuno file di definizione.

Il processo di containerizzazione delle applicazioni con Apptainer è reso semplice grazie alla sua compatibilità con la piattaforma Docker. Docker è la piattaforma open-source più utilizzata al mondo per la containerizzazione delle applicazioni. Il funzionamento e la sintassi dei file di definizione di Apptainer sono molto simili a quelli di Docker.

In questo progetto, i file di definizione per le immagini utilizzano Docker come *bootstrap agent*, ovvero come responsabile per la creazione dell'immagine di base su cui verranno eseguite le istruzioni specificate nel file di definizione. In particolare, tutti i file di definizione sviluppati iniziano con le seguenti istruzioni.

```
1 Bootstrap: docker
2 From: ubuntu:20.04
```

L'istruzione `Bootstrap` definisce l'agente di bootstrap e abilita l'utilizzo dell'istruzione `From`, che serve per definire l'immagine di base del container. Se l'immagine non è presente sul sistema locale, viene scaricata da Apptainer dall'opportuno servizio di hosting di immagini, in questo caso DockerHub. Tutte le immagini create in questo progetto utilizzano come immagine di base `ubuntu:20.04`, poiché entrambe le implementazioni sono state sviluppate sul sistema operativo Ubuntu versione 20.04.

Per comprendere meglio i vari file di definizione sviluppati, è opportuno descrivere il significato di ogni sezione presente in questi file.

- `%files`: In questa sezione vengono trasferiti i file necessari dal sistema host al container.
- `%environment`: Questa sezione definisce le variabili di ambiente che verranno utilizzate durante l'esecuzione o l'installazione dei diversi programmi all'interno del container.
- `%post`: Questa sezione contiene la sequenza di istruzioni che verranno eseguite durante la costruzione dell'immagine mediante il comando `apptainer build`. La sezione `%post` può essere considerata come la parte responsabile dell'installazione di vari strumenti e librerie necessari per l'applicazione containerizzata.

Per eseguire il processo di containerizzazione delle due applicazioni descritte in seguito, è necessario avere Apptainer installato. Si precisa che l'installazione di Apptainer richiede i permessi di amministratore, pertanto non può essere effettuata senza i relativi privilegi da superutente. Dopo l'installazione, tuttavia, è possibile eseguire e costruire immagini anche senza i privilegi da amministratore.

3.1 MPI

Apptainer offre due modalità per la containerizzazione di applicazioni MPI: il modello *Ibrido* e il modello *Bind*. Entrambi i modelli consistono nel trasferimento dei file necessari all'interno del

contenitore e nell'installazione delle librerie di base per l'esecuzione dell'applicazione. Tuttavia, solo il modello ibrido include l'installazione della libreria MPI all'interno del contenitore.

Questo progetto ha utilizzato il modello ibrido perché il modello bind richiede la conoscenza del percorso di installazione di MPI sul dispositivo ospitante. Questo requisito potrebbe rendere l'esecuzione meno pratica per un utente che utilizzi un dispositivo diverso dal proprio.

Detto questo, anche il modello Ibrido presenta due requisiti. Per prima cosa, il sistema operativo ospitante deve avere la libreria MPI correttamente installata. Secondo, è importante che la versione MPI del sistema ospitante sia compatibile con quella installata all'interno del container.

Rispettati tutti i requisiti, la containerizzazione dell'implementazione MPI risulta essere relativamente semplice. Analizziamo quindi il file di definizione sviluppato.

```
4  %files
5      bilateral-filter-mpi.c /opt
6      hpc.h /opt
7      ppm.h /opt
```

In questa prima sezione,vengono caricati i file dell'implementazione MPI all'interno della cartella `/opt` nel container.

```
8  %environment
9      export OMPI_DIR=/opt/ompi
10     export PATH="$OMPI_DIR/bin:$PATH"
11     export LD_LIBRARY_PATH="$OMPI_DIR/lib:
12                             $LD_LIBRARY_PATH"
```

Nella seconda porzione `%environment`, come spiegato precedentemente, vengono definite le varie variabili di ambiente poi utilizzate per l'installazione di MPI all'interno del container. La prima variabile consiste nel percorso di installazione che sarà utilizzato per la librearia MPI, e di conseguenza vengono aggiornate la variabile di `PATH` e `LD_LIBRARY_PATH` rendendo così possibile l'utilizzo di MPI.

```
13 %post
14     apt-get update
15     apt-get install -y wget bash gcc
16     apt-get install -y g++ make
```

Concludendo con la sezione `%post`, le prime tre istruzioni aggiornano e installano le diverse librerie necessarie per l'installazione di MPI e l'esecuzione dell'applicazione.

```
17     export OMPI_DIR=/opt/ompi
18     export OMPI_VERSION=4.1.4
19     export OMPI_URL="https://download.open-mpi.org/
20       release/open-mpi/v4.1/openmpi-$OMPI_VERSION.tar.bz2"
21     mkdir -p /tmp/ompi
22     mkdir -p /opt
23     cd /tmp/ompi
24     wget -O openmpi-$OMPI_VERSION.tar.bz2 $OMPI_URL
25     tar -xjf openmpi-$OMPI_VERSION.tar.bz2
26     cd /tmp/ompi/openmpi-$OMPI_VERSION
27     ./configure --prefix=$OMPI_DIR
28     make -j4 install
```

La seconda sezione di istruzioni esegue l'installazione della libreria MPI scaricando l'archivio di installazione dal sito web

ufficiale di OpenMPI. Per fare ciò, vengono create varie cartelle temporanee e di installazione, concludendo con l'installazione completa di MPI tramite l'istruzione `make install`.

Per l'installazione della libreria OpenMPI esistono diverse alternative, tuttavia il metodo utilizzato in questo file di definizione è stato scelto in quanto corrisponde all'approccio descritto nella documentazione MPI di Apptainer.

```
29      export PATH=$OMPI_DIR/bin:$PATH
30      export LD_LIBRARY_PATH=$OMPI_DIR/lib:
31                             $LD_LIBRARY_PATH
```

Completata l'installazione di MPI, vengono aggiornate le variabili di ambiente per il percorso di installazione della libreria, preparando così l'ambiente per le successive operazioni.

```
32      cd /opt
33      mpicc -std=c99 -Wall -Wpedantic -O2
34          bilateral-filter-mpi.c -o mpi-bilateral -lm
```

Il file di definizione si conclude con la compilazione dell'applicazione del filtro bilaterale tramite il comando `mpicc`, creando così l'eseguibile all'interno del container.

Per l'esecuzione dell'applicazione containerizzata si procede con la creazione del *Singularity Image File* utilizzando il file definizione appena analizzato:

```
apptainer build --fakeroot bilateral-mpi-hybrid.sif \
    bilateral-mpi-hybrid.def
```

Il comando `apptainer build` si occupa di costruire l'immagine del container a partire dal file di definizione. Il flag `--fakeroot` rende possibile la costruzione del container anche

per utenti che non hanno permessi da amministratore. Ottenuto il file immagine `bilateral-mpi-hybrid.sif`, l'applicazione containerizzata può essere eseguita tramite il comando:

```
mpirun -n <num_proc> apptainer exec  \
    bilateral-mpi-hybrid.sif /opt/mpi-bilater \
    <img> <radius> <sigma_c> <sigma_s>
```

Il comando richiama MPI dal sistema ospitante con il comando `mpirun`, definendo il numero di processori da utilizzare. In seguito inizia il segmento della containerizzazione, tramite il comando `apptainer exec`. Tale comando manda in esecuzione una determinata istruzione all'interno del container definito dall'immagine `bilateral-mpi-hybrid.sif`. L'istruzione consiste nell'esecuzione dell'applicazione precedentemente compilata all'interno del container, passando tutti i vari parametri necessari (Raffigurazione containerizzazione MPI in figura 3.1)

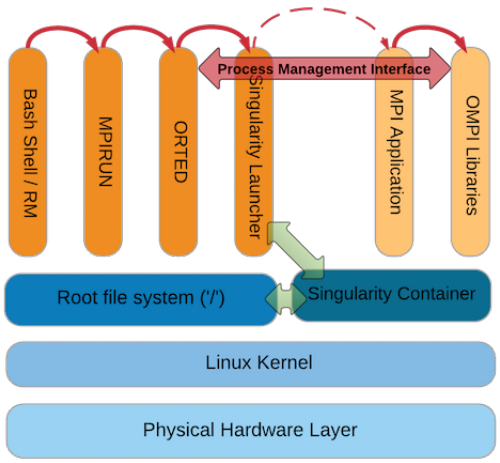

Figura 3.1: Esecuzione di applicazioni MPI in Apptainer (Singularity).
fonte: `https://tin6150.github.io/psg/blogger_container_hpc.html`

3.2 CUDA

Per la containerizzazione di applicazioni CUDA, Apptainer prevede un unico metodo di containerizzazione (il secondo metodo è ancora in stato sperimentale).

Per l'esecuzione dell'applicazione è necessaria la presenza di una scheda video NVIDIA nel dispositivo host, con i relativi driver installati. Un metodo per verificare la presenza di una scheda video valida e dei rispettivi driver è con il seguente comando:

```
nvidia-smi
```

Un altro requisito da rispettare per l'esecuzione dell'applicazione è l'installazione della libreria CUDA anche sul dispositivo ospitante. Anche in questo caso il file immagine per il container è stato ottenuto sviluppando un opportuno file di definizione.

```
4  %files
5      bilateral-filter-cuda.cu /opt
6      hpc.h /opt
7      ppm.h /opt
```

Similmente all'implementazione containerizzata MPI, la prima sezione si limita a trasferire i file dell'implementazione CUDA all'interno del container.

```
8  %post
9      apt-get update
10     apt-get upgrade -y
11
12     apt-get install -y gcc
13     apt-get install -y build-essential
```

```
14
15      apt-get install dialog apt-utils -y
16      echo 'debconf debconf/frontend select
17          Noninteractive' | debconf-set-selections
18      apt-get install -y -q
```

Nella successiva sezione `%post`, vengono aggiornate e installate alcune librerie di base per il corretto funzionamento dell'applicazione all'interno del container. Un'altra operazione fondamentale è quella eseguita nelle righe 15, 16, 17, queste istruzioni assicurano una corretta installazione di CUDA.

```
8       apt-get install -y nvidia-cuda-toolkit
9
10      cd /opt
11      nvcc bilateral-filter-cuda.cu -o cuda-bilateral
```

Il file definizione finisce con l'installazione di CUDA e la compilazione dell'eseguibile `cuda-bilateral` all'interno del container. A partire dal file di definizione appena analizzato è possibile ottenere il file immagine `bilateral-cuda.sif` con il comando:

```
apptainer build --fakeroot bilateral-cuda.sif \
   bilateral-cuda.def
```

L'esecuzione dell'implementazione containerizzat CUDA risulta più immediata rispetto a quella MPI.

```
apptainer run --nv bilateral-cuda.sif \
   /opt/cuda-bilateral <img> <sigma_c> <sigma_s>
```

Tramite il comando `apptainer run` si manda in esecuzione l'eseguibile all'interno del container, specificando i vari parametri richiesti. L'esecuzione dell'applicativo sulla scheda video del sistema ospitante è reso possibile con il flag `--nv`.

Capitolo 4

Valutazione delle prestazioni

In considerazione dell'obiettivo del progetto, l'analisi delle prestazioni mira a esaminare l'effetto della containerizzazione sui tempi di esecuzione delle implementazioni sviluppate. L'obiettivo è quello di valutare l'impatto che la containerizzazione ha sui tempi di esecuzione, al fine di fornire una valutazione accurata della tecnologia utilizzata.

4.1 MPI

Prima di analizzare i dati presentati nella tabella 4.1, è necessario definire cosa rappresentano i tempi di esecuzione e come sono stati misurati.

Nel contesto dei tempi di esecuzione nativi, il tempo totale impiegato dal programma comprende diverse fasi: la verifica e la lettura dei parametri e dell'immagine in ingresso, l'applicazione del filtro Bilaterale e la scrittura dell'immagine filtrata. Per quanto riguarda i tempi di esecuzione dell'applicazione containerizzata con Apptainer, i tempi misurati comprendono non solo il tempo impiegato per eseguire le stesse operazioni della

versione nativa, ma anche il tempo necessario per la creazione e la terminazione del container.

Tutti i tempi di esecuzione elencati e analizzati in seguito sono stati ottenuti mediante l'utilizzo del comando `time` all'interno di un terminale Linux. Tale comando è utilizzato per misurare i tempi di esecuzione di un determinato comando o programma. Al fine di ottenere tempi di esecuzione più affidabili, ciascun tempo è stato calcolato come media di 5 esecuzioni. Questo approccio è stato adottato per minimizzare eventuali fluttuazioni e fornire valori più precisi e rappresentativi dei tempi di esecuzione effettivi.

Cores	Nativo	Apptainer
1	10,684	10,86
2	5,723	5,813
3	4,037	4,122
4	3,192	3,273
5	2,649	2,731
6	2,314	2,39

Tabella 4.1: Tempi di esecuzione Implementazione MPI.

La tabella 4.1 riporta i tempi di esecuzione dell'implementazione in secondi, sia in modalità nativa che containerizzata mediante Apptainer, insieme al numero di processori utilizzati (Core) per ciascuna delle due modalità di esecuzione. Per garantire la comparabilità dei risultati e una valutazione accurata delle prestazioni dell'implementazione in entrambe le modalità, si è utilizzata la stessa immagine di ingresso con risoluzione 768×768 pixel e gli stessi parametri di elaborazione, ovvero un `radius`=10 e valori di σ_c e σ_s pari a 15.

Al fine di evidenziare la somiglianza nei tempi di esecuzione, nel grafico rappresentato nella figura 4.1 sono state utilizzate delle barre per confrontare i tempi di esecuzione in relazione all'aumentare del numero di processori utilizzati.

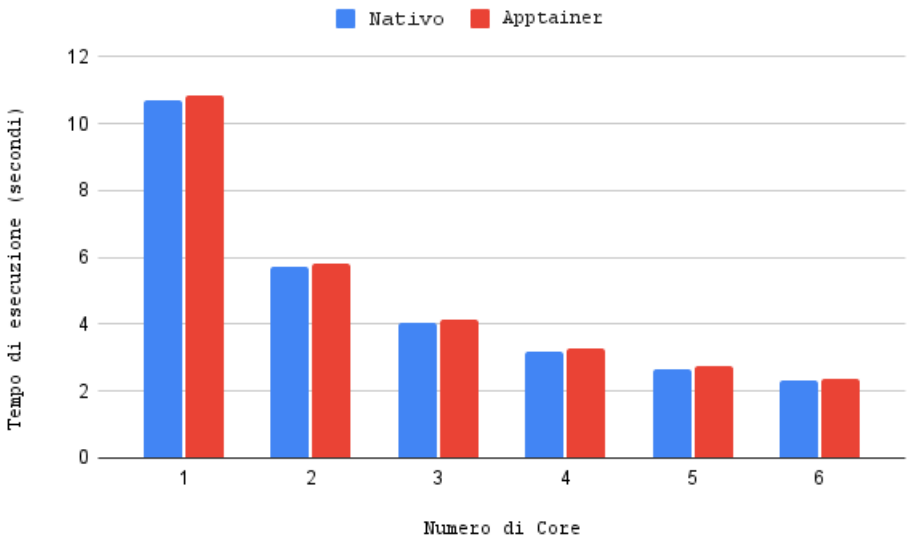

Figura 4.1: Grafico a barre prestazioni MPI nativo e containerizzato.

Nonostante la figura 4.1 possa fornire un'indicazione del livello di rallentamento causato dalla containerizzazione dell'implementazione, è opportuno analizzare più attentamente tale variazione delle prestazioni mediante il rapporto tra il tempo di esecuzione in Apptainer e quello nativo mostrato in figura 4.2.

Il rapporto tra il tempo di esecuzione dell'applicazione in Apptainer e in esecuzione nativa si ottiene tramite la semplice divisione delle due quantità. Questo rapporto sarà sempre maggiore o uguale a 1, poiché il tempo di esecuzione in Apptainer non può essere inferiore a quello in esecuzione nativa.

Più il valore del rapporto si avvicina a 1, più il tempo di esecuzione di Apptainer sarà simile a quello della versione nativa. Viceversa, più il valore del rapporto si discosta da 1, più le prestazioni di Apptainer risulteranno peggiori rispetto alla versione nativa.

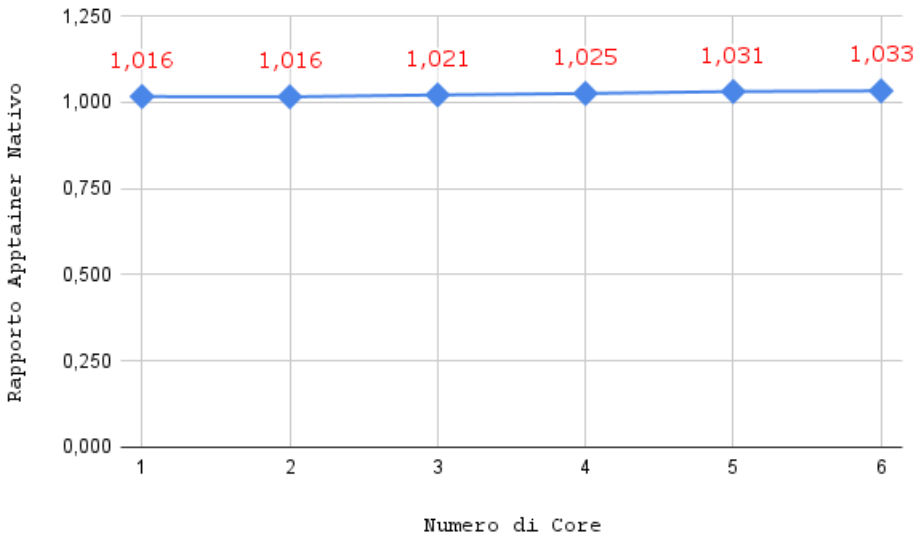

Figura 4.2: Rapporto tempo di esecuzione Apptainer e Nativo MPI.

Dall'analisi dei dati raccolti, in particolare dai rapporti dei tempi di esecuzione mostrati nella figura 4.2, possiamo dedurre due informazioni principali riguardanti le prestazioni dell'implementazione MPI containerizzata.

In primo luogo, è possibile notare che all'aumentare del numero di core utilizzati, il rapporto tra i tempi di esecuzione di Apptainer e quelli nativi rimane sempre molto vicino a 1. Questo suggerisce che l'impatto negativo di Apptainer sulle prestazioni risulta essere relativamente ridotto.

In secondo luogo, la stabilità del rapporto dei tempi di esecuzione indica che l'impatto negativo sulle prestazioni non varia in base al numero di core utilizzati. In altre parole, l'effetto negativo sulle prestazioni non aumenta o diminuisce significativamente in relazione all'uso di un maggior o minore numero di core.

4.2 CUDA

Per misurare i tempi di esecuzione dell'implementazione CUDA, è stato utilizzato il comando `time`, allo stesso modo dell'implementazione MPI. Per ottenere misure affidabili, ciascun tempo è stato calcolato come media di 5 esecuzioni.

In modo analogo ai tempi di esecuzione per l'implementazione MPI, anche i tempi di esecuzione dell'implementazione nativa CUDA tengono conto del tempo necessario per eseguire le operazioni di lettura e verifica dell'immagine in ingresso, l'applicazione del filtro e la scrittura dell'immagine filtrata.

I tempi di esecuzione dell'implementazione containerizzata, invece, comprendono le stesse operazioni insieme alla creazione e terminazione del container.

I tempi di esecuzione riportati nella tabella 4.2 sono stati ottenuti aumentando la risoluzione dell'immagine in ingresso al programma. Questo è stato fatto al fine di garantire la comparabilità dei tempi di esecuzione, utilizzando gli stessi parametri di configurazione: `radius`=10 e valori di σ_c e σ_s pari a 15.

Risoluzione	Nativo	Apptainer
256 × 256	0,196	0,517
512 × 512	0,314	0,611
768 × 768	0,482	0,744
912 × 912	0,674	0,945
1024 × 1024	2,363	2,653
1920 × 1920	10,119	10,382

Tabella 4.2: Tempi di esecuzione Implementazione CUDA.

Per una più accurata valutazione della similarità dei tempi di esecuzione dell'implementazione CUDA, si adotta un approccio simile a quello utilizzato per l'implementazione MPI. I tempi di esecuzione sono rappresentati in un grafico a barre, mostrato nella figura 4.3, per permettere una comparazione tra i risultati ottenuti a diverse risoluzioni di immagini.

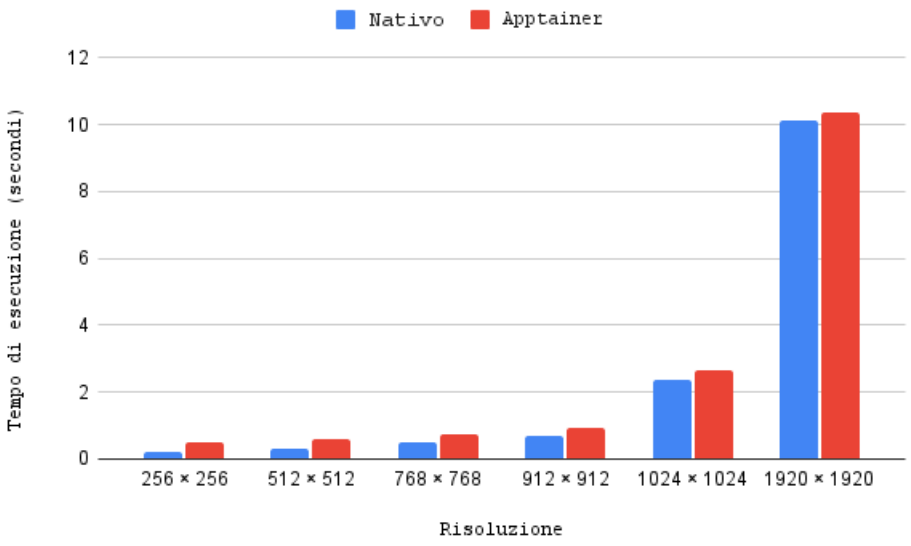

Figura 4.3: Grafico a barre prestazioni CUDA nativo e containerizzato.

Analizziamo più attentamente la variazione delle prestazioni per l'implementazione CUDA attraverso il rapporto tra il tempo di esecuzione in Apptainer e quello nativo, come mostrato nella figura 4.4.

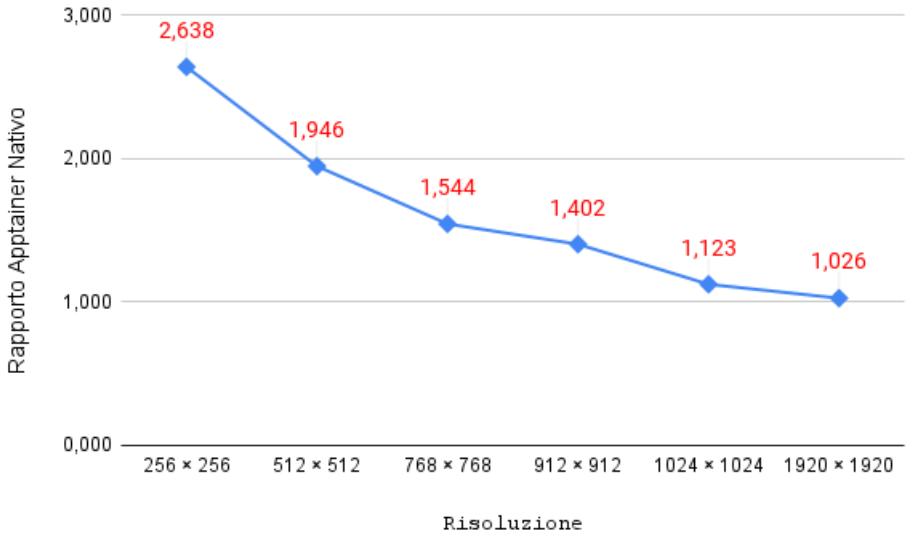

Figura 4.4: Rapporto tempo di esecuzione Apptainer e Nativo CUDA.

Si conclude l'analisi dei tempi di esecuzione per questa implementazione con un commento relativo alle informazioni rappresentate dai rapporti in figura 4.4. L'andamento decrescente dei rapporti tra i tempi di esecuzione in Apptainer e nativo può essere spiegato facilmente considerando la risoluzione delle immagini in ingresso.

L'impatto negativo della containerizzazione Apptainer sul tempo di esecuzione risulta maggiore su immagini di bassa risoluzione. Questo è dovuto al fatto che l'applicazione del filtro bilaterale con l'implementazione CUDA su immagini di bassa risoluzione richiede un tempo di esecuzione molto basso, quin-

di il rallentamento causato dalla containerizzazione sarà più evidente.

Pertanto, si sconsiglia di utilizzare la containerizzazione Apptainer per applicazioni con tempi di esecuzione brevi. Tuttavia, i rapporti molto vicini a 1 ottenuti con una risoluzione maggiore di 1024 indicano che la containerizzazione con Apptainer non è troppo dannosa per le applicazioni con tempi di esecuzione superiori a 1 secondo.

Capitolo 5

Conclusioni

Il presente progetto ha condotto un'analisi del processo di containerizzazione delle applicazioni HPC attraverso l'uso di Apptainer. In particolare, si sono esaminate le diverse fasi richieste dal processo di containerizzazione delle applicazioni utilizzando questo sistema, nonché le relative prestazioni. Si può affermare che tale processo comprende le tradizionali operazioni dell'HPC per la produzione di implementazioni eseguibili in parallelo, cui viene aggiunta una fase di containerizzazione delle implementazioni. Quest'ultima fase, grazie alla facilità d'uso di Apptainer e al suo supporto per altri sistemi di containerizzazione, richiede solamente una conoscenza basilare dei principi della containerizzazione.

Dall'analisi delle prestazioni di entrambe le implementazioni, si evince che Apptainer garantisce prestazioni vicine a quelle ottenute in modalità nativa. L'utilizzo di Apptainer, pertanto, non comporta eccessivi rallentamenti delle applicazioni.

In conclusione, Apptainer si presenta come un valido metodo per l'esecuzione di applicazioni containerizzate su un sistema HPC avanzato, come dimostra il fatto che diversi HPC Cluster hanno già adottato questo sistema di containerizzazione [1].

Bibliografia

[1] Apptainer website. Accessed on 23 February, 2023. URL: `https://apptainer.org/`.

[2] Repository codice progetto. Accessed on 26 February, 2023. URL: `https://github.com/pietropezzi/Containerized-HPC_Thesis`.

[3] Manhal Elfadil, Abd Latiff Shafie, and Ismail Isnin. Distributed memory and shared distributed memory architecture for implementing local sequences alignment: A survey. *International Journal of Computer Science and Telecommunications*, 5:1–8, 2014.

[4] Abel Paz-Gallardo and Antonio Plaza. A new morphological anomaly detection algorithm for hyperspectral images and its gpu implementation. *Proceedings of SPIE - The International Society for Optical Engineering*, 8157, 2011.

[5] C. Tomasi and R. Manduchi. Bilateral filtering for gray and color images. *Sixth International Conference on Computer Vision (IEEE Cat. No.98CH36271)*, pages 839–846, 1998. URL: `https://ieeexplore.ieee.org/document/710815`.

[6] Naweiluo Zhou, Yiannis Georgiou, Marcin Pospieszny, Li Zhong, Huan Zhou, Christoph Niethammer, Branislav

Pejak, Oskar Marko, and Dennis Hoppe. Container orchestration on hpc systems through kubernetes. *Journal of Cloud Computing*, 10, February 2021. URL: `https://doi.org/10.1186/s13677-021-00231-z`.