Appunti Didattici

Future Computing Architectures and Programming Paradigms

versione 1.0.0

Professore:
Prof. Raffaele Montella

Autori:
Francesco Grimaldi
Paolo Palmiero

Anno accademico 2023/2024

Indice

1 FPP: Parallel Programming Models

(Montella)

1.1 Tassonomia di Flynn

La tassonomia di Flynn è una classificazione delle architetture di computer. Secondo tale modello possiamo avere le seguenti architetture:

1. **SISD** (Single Input Stream Single Data Stream): abbiamo un singolo flusso di istruzioni eseguito da un'unica CPU. In generale non vi è parallelismo: le operazioni vengono eseguite sequenzialmente, su un dato alla volta. Il parallelismo, se c'è, si ottiene con più core e con tecniche di accelerazione del ciclo, ma non si rientra più in tale categoria. Un esempio è la classica macchina di Von Neumann e tutti i computer venduti prima del 2010;

2. **SIMD** (Single Input Stream Multiple Data Stream): esecuzione in parallelo della stessa istruzione su insiemi di dati differenti. Per esempio array processor e vector supercomputer;

3. **MISD** (Multiple Input Stream Single Data Stream): istruzioni distinte lavorano contemporaneamente sugli stessi dati. Non ci sono esempi rilevanti;

4. **MIMD** (Multiple Input Stream Multiple Data Stream): istruzioni distinte vengono eseguite su flussi di dati distinti. Sistemi *multiprocessore* e *multicomputer*.

1.2 MIMD vs SIMD

Task parallelism MIMD. E' una forma di parallelizzazione su più processori in ambienti di elaborazione parallela. Il task parallelism si concentra sulla distribuzione dei task, eseguiti simultaneamente da processi o thread, su diversi processori. Pertanto è un modello fork-join[1] con un parallelismo thread-level che utilizza o un paradigma shared memory o message passing. La sincornizzazione deve essere esplicita.

Data parallelism SIMD. E' una forma di parallelizzazione su più processori in ambienti di elaborazione parallela. Si concentra sulla distribuzione dei dati tra diversi nodi, che operano sui dati in parallelo. Può essere applicata a strutture di dati regolari come array e matrici, lavorando su ciascun elemento in parallelo. Quindi più processori che operano su segmenti di insiemi di dati. Si

[1] Il modello fork-join è un modo di impostare ed eseguire programmi paralleli in modo che l'esecuzione si dirami in parallelo in punti designati del programma, per "unirsi" (join) in un punto successivo e riprendere l'esecuzione sequenziale. Le sezioni parallele possono fare dei fork in modo ricorsivo fino a raggiungere una certa granularità di task.

può avere su modelli SIMD con macchine vettoriali e macchine con pipeline. La comunicazione avviene attraverso la shared memory o spazi di indirizzamento logici condivisi. La sincronizzazione è implicita.

1.3 Parallel Programming Models

Facciamo qui riferimento a cinque tipologie di modelli di programmazione: multiprogramming model, shared address space programming, message passing programming, hybrid systems e bulk synchronous processing.

Multiprogramming Model. Come suggerisce il nome, abbiamo più programmi (o task) attivi allo stesso momento e totalmente indipendenti gli uni dagli altri. Non c'è né comunicazione né sincronizzazione a livello di programma e l'idea generale è assegnare la CPU ad altri processi quando l'attuale processo potrebbe non essere terminato. I *vantaggi* evidenti sono che all'utente sembra che stia eseguendo diverse applicazioni sulla stessa CPU anche se quest'ultima esegue un processo per volta e, in secondo luogo, l'utilizzo della CPU è ottimizzato. Tutti i sistemi operativi attuali utilizzano il multiprogramming. Gli *svantaggi* sono legati al fatto che sono necessari algoritmi di scheduling; se sono presenti tanti task, quelli più longevi verrano terminati ancor più tardi; è necessaria una gestione della **memoria** perché tutti i task sono memorizzati nella RAM.

Più nel dettaglio il multiprogramming è un modello *shared memory task parallel*, tipico dei sistemi MIMD, basato sui thread: ogni processo può essere caratterizzato da più thread. Si opera collettivamente su un insieme di dati condivisi: ogni thread ha le proprie variabili private, ossia dati sullo stato del thread, variabili locali presenti sulla stack di runtime. I thread si coordinano esplicitamente tramite operazioni di sincronizzazione su variabili condivise, che comportano: creazione e unione dei thread; scrivere e leggere flag; usare lock e semafori, per esempio per garantire la mutua esclusione.

Occorre fare una differenza sostanziale tra UMA, NUMA e DSM.

- **UMA (Uniform Memory Access)**: macchina shared memory in cui ogni processore ha un accesso uniforme alla memoria. Si parla di *symmetric multiprocessors* (SMP) ossia computer multiprocessore in cui due o più identici processori hanno accesso a una singola *shared main memory*. Non c'è una memoria locale o privata, pertanto tutte le variabili locali sono messe nella shared memory. E' illustrato in Figura 1.

- **NUMA (Not Uniform Memory Access**): il tempo di accesso alla memoria dipende dalla posizione dei dati rispetto al processore. In NUMA ogni processore può accedere molto più velocemente alla sua memoria locale rispetto all'accesso a una memoria non locale. In linea generale uno o più CPU hanno una propria memoria locale, diversa da quella di un'altra CPU o gruppo di CPU come illustrato in Figura 2.

- **DSM (Distributed Shared Memory**): è un'architettura di memorie in cui memoria fisicamente separate possono essere indirizzate come un singolo spazio di memoria condiviso. Genericamente l'accesso è più lento rispetto all'accesso a memoria condivise non distribuite, è richiesta infatti una comunicazione. I vantaggi sono che scala bene con un grande numero di nodi; è più economico rispetto a sistemi multiprocessore; fornisce una grande memoria virtuale; rende i programmi più portabili. E' illustrato in Figura 3.

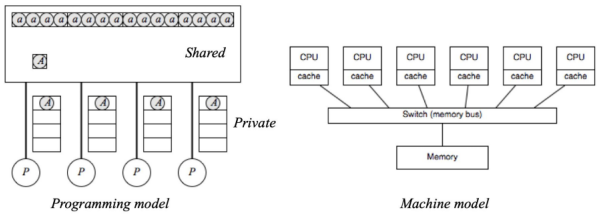

Figura 1: UMA.

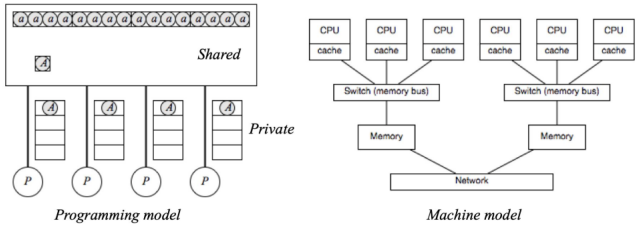

Figura 2: NUMA.

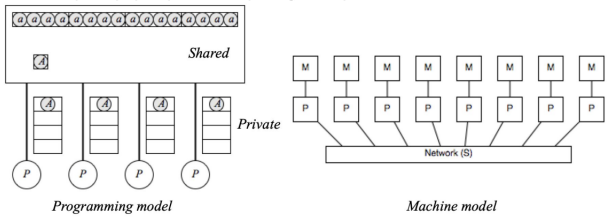

Figura 3: DSM.

Shared Address Space Programming. E' un paradigma di programmazione ove i task operano e comunicano attraverso dati condivisi, i.e. usano una memoria condivisa. Un esempio è OpenMP. E' un modello *data parallel* in cui

vi è un singolo thread di controllo che gestisce operazioni parallele, le quali sono applicate a un segmento specifico di una struttura dati, come un array. La comunicazione e la sincronizzazione sono implicite.

Un esempio è la programmazione data parallel con una vector machine in cui un'istruzione viene eseguita su più elementi di dati, in genere in modo pipeline.

Un altro esempio è la programmazione data parallel con una macchina SIMD in cui vi è un numero elevato di processori semplici a cui un processore di controllo impartisce istruzioni. Ogni processore esegue la stessa istruzione (in lock-step). I processori vengono disattivati selettivamente per il flusso di controllo del programma.

Message Passing Programming. In questo modello un programma è costituito da un insieme di processi *nominati*: il singolo processo ha un thread di controllo e una memoria locale con uno spazio di indirizzamento locale. I processi possono tra loro comunicare attraverso trasferimenti di dati espliciti, in particolare vengono scambiati messaggi tra una sorgente e una destinazione che sono entrami processori nominati (P0, P1, ...) o nodi di calcolo. Per lo scambio di messaggi possono essere usate librerie come per esempio MPI.

Più nel dettaglio, ogni nodo ha un'interfaccia di rete tramite la quale è possibile effettuare la comunicazione e la sincronizzazione. La latenza dei messaggi e la bandwidth dipende fortemente dalla topologia di rete e dagli algoritmi di routing. Una rappresentazione con una possibile topologia di rete è in Figura 4.

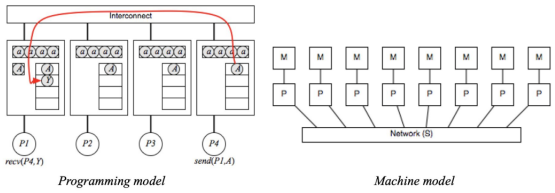

Figura 4: Message passing programming.

Hybrid Systems. Si fa riferimento a cluster SMP[2]. In questo caso, il modello di programmazione offre tre scelte:

1. **"Flat" system**: viene usato il paradigma message passing anche con un'architettura SMP. I vantaggi sono la portabilità e la facilità di programmazione. Lo svantaggio principale è l'ignorare totalmente la gerarchia di memoria SMP e i vantaggi forniti dallo spazio di indirizzamento condiviso UMA.

2. **Programma in due layer**: viene sfruttato sia la shared memory programming che il message passing. Si raggiungono migliori prestazioni a scapito di una maggiore difficoltà di programmazione.

[2]SMP (symmetric multiprocessing) si riferisce an'architettura in cui più processori identici sono interconnessi a una singola memoria principale condivisa.

3. **Programma in tre layer**: SIMD per ogni core, shared memory programming tra core su uno stesso nodo SIMD e message passing tra i nodi. Un esempio è illustrato in Figura 5.

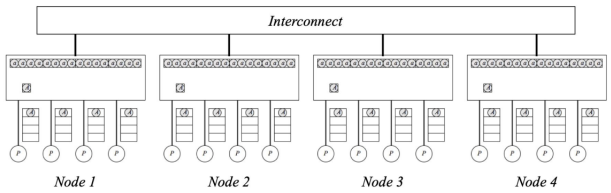

Figura 5: Programma in tre layer.

Bulk Synchronous Processing. Il programma parallelo è composto dai cosiddetti *superstep*. Ogni superstep è a sua volta costituito da tre fasi:

1. **Compute phase**: i processi operano sui dati locali, compresi accessi in lettura alla shared memory su un'architettura SMP.

2. **Communication phase**: tutti i processi cooperano sullo scambio di dati o sulla reduction dei dati globali.

3. **Barrier synchronization**.

Ci si assicura che le fasi di calcolo e comunicazione siano completate prima del successivo superstep.

Con questo modello si garantisce semplicità della programmazione parallela dei dati, senza le restrizioni.

2 FPP: Code Offloading

Il **code offloading** è il trasferimento di task computazionali ad alta intensità di risorse a un processore separato, come un acceleratore hardware, o a una piattaforma esterna, come un cluster, una griglia o un cloud.

2.1 NVIDIA GPU Technology

Il paradigma di programmazione prevede la suddivisione del codice in due parti: host code (CPU) e device code (GPU). L'architettura hardware della GPU richiede un program flow altamente omogeneo (senza if). Il PCIe rappresenta un bottleneck per la comunicazione dei dati tra host e device ed è necessario sovrapporre le fasi di computazione con quelle di comunicazione.

I linguaggi di programmazione usati sono CUDA e OpenCL e il codice si divide nel programma host in C, eseguito sulla CPU, e i kernel del device sempre scritti in C ma lanciati sul device. Sono a disposizione tool per il debuggin, per il system monitoring, per il profiling e altri. I linguaggi sono fortemente di alto livello, al pari di OpenMP.

2.2 GPU e CPU

Una **GPU** (Graphics Processing Unit) è un sistema di elaborazione che nasce per soddisfare le esigenze associate alla computazione grafica. Esso è sostanzialmente un coprocessore che entra in aiuto delle CPU che non sono in grado di eseguire computazioni massive. Col tempo ci si rese conto che il potenziale delle GPU potesse essere sfruttato su un grande campo di applicazionie e quindi si sono guadagnate il nome di **GPGPU (General-Purpose GPU**).

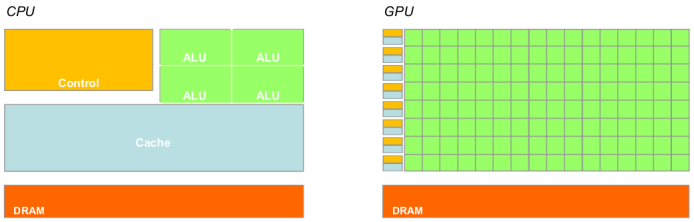

Figura 6: CPU e GPU.

Le CPU sono caratterizzate principalmente da una cache di livello due, come si nota nella Figura 6 che occupa buona parte del chip; da una unità di controllo e da multiple unità di elaborazione (ALU). Nella cache sono memorizzati insiemi di dati in base al principio di località spaziale e temporale e questo permette un accesso ai dati stessi molto rapido. Possiamo quindi affermare che le CPU sono ottimizzate per avere una *bassa latenza* (tempo di accesso ai dati). Questo però comporta prestazioni più basse in termini di elaborazione e quindi il *throughput si riduce*. Per le GPU il discorso è del tutto opposto: è presente una notevole

duplicazione dell'hardware dedicato all'elaborazione permettendo calcolo massimo, a discapito di una memoria cache di livello due molto più piccola. Pertanto le GPU presentano una *elevata latenza* e *alto throughput*.

2.3 CUDA

CUDA (Compute Unified Device Architecture) è una piattaforma di calcolo parallelo general purpose e un modello di programmazione che facilita la programmazione delle GPU NVIDIA, che fornisce:

- Un nuovo paradigma di programmazione multi-thread gerarchico;
- Un nuovo set di istruzioni per l'architettura chiamato PTX (Parallel Thread eXecution);
- Un piccolo insieme di estensioni della sintassi dei linguaggi di programmazione di livello superiore (C, Fortran) per esprimere il parallelismo dei thread in un familiar programming environment;
- Una raccolta completa di strumenti di sviluppo per la compilazione, il debug e il profiling dei programmi CUDA.

2.3.1 CUDA Programming Model

La GPU è vista come un coprocessore ausiliario con un proprio spazio di memoria. Sulla GPU possono essere eseguite porzioni di programma ad alta intesità di calcolo e data-parallel. Ogni porzione di calcolo data-parallel può essere isolata in una funzione, chiamata kernel CUDA, che viene eseguita sulla GPU e i kernel CUDA vengono eseguiti da molti thread diversi in parallelo: ogni thread può calcolare diversi elementi di dati in modo indipendente. Il parallelismo della GPU è molto vicino al paradigma SPMD (Single Program Multiple Data), in paricolare Single Instruction Multiple Threads (SIMT) secondo la definizione di Nvidia.

2.3.2 CUDA Execution Model

La CPU e la GPU lavorano in concomitanza e sono connessi da BUS ad altre prestazioni (PCIe). Il flusso di elaborazione segue i seguenti passi:

1. copia dei dati dalla memoria della CPU alla memoria della GPU[3];
2. carica del programma nella GPU e esecuzione massiva;
3. scrittura dei risultati nella memoria della GPU;
4. copia dei risultati dalla memoria della GPU alla memoria della CPU.

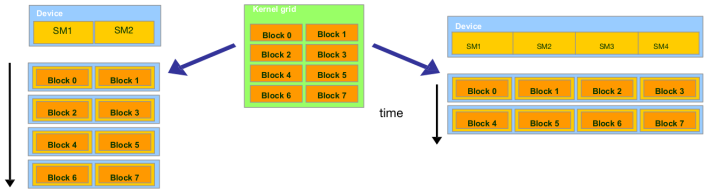

Figura 7: Scalabilità trasparente.

Nelle GPU più recenti questo flusso avviene in maniera trasparente al programmatore.

Quando un CUDA kernel è invocato ogni thread block è assegnato a uno Streaming Multiprocessor[4] (SM) in modalità round-robin: a ogni SM può essere assegnato un numero massimo di blocchi, a seconda della generazione dell'hardware e del numero di risorse di cui ogni blocco ha bisogno per essere eseguito (registri, memoria condivisa, ecc.). Il sistema di runtime mantiene un elenco di blocchi che devono essere eseguiti e assegna nuovi blocchi agli SM man mano che questi completano l'esecuzione dei blocchi precedentemente assegnati. Una volta che un blocco è assegnato a un SM, rimane su tale SM fino a quando il lavoro di tutti i thread del blocco non è stato completato L'esecuzione di ciascun blocco è indipendente dagli altri (non è possibile alcuna sincronizzazione tra di essi). I thread di ogni blocco sono partizionati in warp di 32 thread ciascuno, in modo da mappare ogni thread con un unico indice di thread consecutivo nel blocco, a partire dall'indice 0. Lo scheduler seleziona per l'esecuzione un warp da uno dei blocchi residenti in ogni SM. Un warp esegue un'istruzione comune alla volta: ogni CUDA core si occupa di un thread nel warp in piena efficienza quando tutti i thread sono d'accordo sul loro percorso di esecuzione.

Il sistema di runtime CUDA può eseguire i blocchi in qualsiasi ordine rispetto agli altri. Questa flessibilità consente di eseguire lo stesso codice applicativo su hardware con un numero diverso di SM (Figura 7).

2.3.3 Heterogeneous High Performance Programming Framework

OpenCL e CUDA, i due principali framework di programmazione per il GPU Computing, si sono contesi la scena nella comunità degli sviluppatori negli ultimi anni. Fino a poco tempo fa, CUDA ha attirato la maggior parte dell'attenzione degli sviluppatori, soprattutto nell'ambito del calcolo ad alte prestazioni. Ma il software OpenCL è ora maturato al punto da indurre i professionisti dell'HPC a riconsiderarlo.

Sia OpenCL che CUDA forniscono un modello general-purpose per il parallelismo dei dati e l'accesso di basso livello all'hardware, ma solo OpenCL offre un framework aperto e standard di settore. Per questo motivo, ha ottenuto il

[3]Le GPU presentano una gerarchia di memorie diversa da quella classica. In questo caso la copia avviene nella Global Memory

[4]Gli SM sono i componenti della GPU che effettuano la reale computazione.

supporto di quasi tutti i produttori di processori, tra cui AMD, Intel e NVIDIA, oltre ad altri che operano nei mercati del mobile e dell'embedded computing. Di conseguenza, le applicazioni sviluppate in OpenCL sono ora portabili su una varietà di GPU e CPU.

Una piattaforma moderna di calcolo include almeno: una o più CPU, una o più GPU, processori DSP[5], acceleratori. Pertanto OpenCL consente ai programmatori di scrivere un singolo programma portabile che utilizza TUTTE le risorse nella piattaforma eterogenea.

2.4 OpenCL

OpenCL è la controparte non proprietaria di CUDA. Supporta le GPU AMD, CPU, MIC, FPGA e altro ancora, quindi è estremamente portabile. Come CUDA, però, è molto di basso livello e richiede buone skill di programmazione per essere usato.

2.4.1 OpenCL Platform Model

L'OpenCL platform model è composto da un host e uno o più OpenCL device. Ogni device è composto da una o più compute unit (CU) e ognuna a loro volta è divisa in uno o più processing element (PE) (Figura 8). La memoria, invece, è divisa in host memory e device memory.

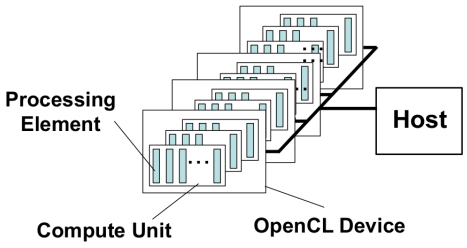

Figura 8: OpenCL platform model.

Le CPU sono trattate come un singolo OpenCL device: una CU per core; una PE per CU o, se le PE sono mappate su SIMD lanes, vi sono n PE per CU.

Ogni GPU, invece, è un OpelCL device distinto: un CU per Streaming Multiprocessor.

E' possibile usare la CPU e tutte le GPU concorrentemente con OpenCL.

L'obiettivo principale di OpenCL è l'estrema portabilità, quindi "espone" tutto e quindi è piuttosto prolisso. Il vantaggio è che il codice dell'host è lo stesso per ogni applicazione e quindi si rimedia in parte allo svantaggio illustrato. In più è possibile impacchettare le API più usate in funzioni che possono essere riutilizzate.

[5]Il processore di segnale digitale (DSP) è un processore dedicato e ottimizzato per eseguire in maniera estremamente efficiente sequenze di istruzioni ricorrenti.

Una differenza tra CUDA e OpenCL è che il primo inizializza automaticamente la GPU e se sono necessarie operazioni più complesse (come l'uso di più device) è possibile farle manualmente. OpenCL invece richiede sempre una inizializzazione esplicita del device in quanto può essere eseguito non solo su GPU NVIDIA e quindi è necessario specificare che device usare.

2.4.2 Shared Memory

Un device può fare uso della *shared memory*: i thread di uno stesso blocco possono cooperare usando la shared memory per condividere dati (Figura 9). Un thread può usare i dati che sono stati già recuperati dalla global memory e posti nella shared memory da un altro thread dello stesso blocco senza ulteriori (lenti) accessi alla global memory.

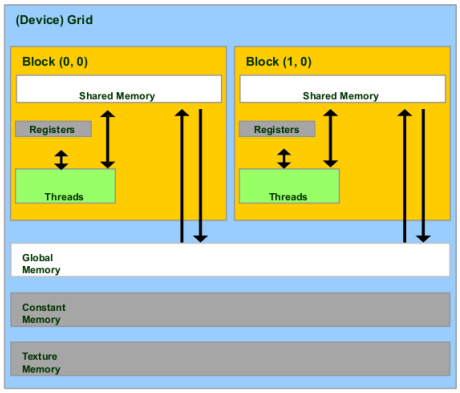

Figura 9: Uso della shared memory.

Un tipico uso della shared memory è il seguente:

1. dichiarare un buffer nella shared memory;
2. caricare i dati nel buffer;
3. sincronizzare i thread del blocco, in modo da essere certi che tutti i dati necessari siano nel buffer;
4. effettuare le operazioni sui dati;
5. sincronizzare i thread in modo da essere certi che tutte le operazioni siano state eseguite correttamente;
6. scrivere i risultati sulla global memory.

3 FPP: Computational Malleability e FlexMPI

3.0.1 Categorie di Job

I job paralleli possono essere classificati in cinque categorie basate sulla flessibilità delle risorse usate. Si parla pertanto di job: *rigid*, *moldable*, *evolving*, *malleable* e *adaptive*.

Rigid job. E' il tipo di job più comune negli ambienti cluster di oggi. Un rigid job viene inviato da un sistema batch[6] che richiede un numero fisso di nodi necessari per eseguire il programma parallelo e il numero di nodi non è modificabile dopo l'invio del job e durante la sua esecuzione (Figura 10). La maggior parte delle applicazioni parallele sono di natura rigida per vari motivi, il più comune dei quali è l'algoritmo utilizzato nell'applicazione stessa che può essere eseguito solo su un numero fisso di nodi. Ad esempio, la decomposizione di un problema di determinate dimensioni di un'applicazione può essere adatta solo a un numero specifico di nodi. Un rigid job definisce anche un tempo di esecuzione dell'applicazione che non può essere superato. Esistono molti metodi per la schedulazione dei rigid job con l'obiettivo di migliorare il throughput e i tempi di risposta, come gli algoritmi best-fit e il backfilling.

Figura 10: Rigid Job.

Moldable job. E' simile a un rigid job, tranne per il fatto che il sistema batch può modificare i requisiti delle risorse dopo l'invio del job, ma prima dell'avvio (Figura 11). Il sistema batch può scegliere di eseguire il job con un numero di risorse inferiore o superiore rispetto al requisito principale specificato, ad esempio per mappare il job sui nodi inattivi attualmente disponibili e migliorare le prestazioni. Pertanto, i moldable job vengono inviati specificando un intervallo di numero di nodi accettabili (minimo e massimo) per il job e un tempo di esecuzione definito per ogni specifica. Il job è poi eseguito almeno quando le risorse minime sono disponibili. Per esempio i job MPI sono moldable se l'applicazione è in grado di scomporre il problema in base al numero di risorse disponibili per l'esecuzione.

[6] Un **sistema batch** prevede l'esecuzione di più istruzioni o programmi (chiamati anche **job**) senza che sia necessario l'intervento di un operatore umano.

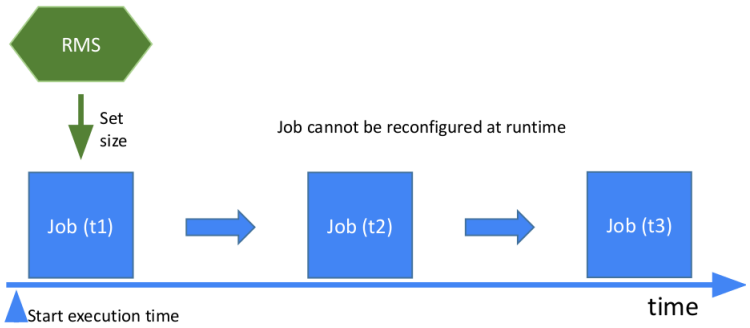

Figura 11: Moldable Job.

Evolving job. A differenza dei rigid e moldable job, un evolving job può cambiare il suo set di allocazione delle risorse durante l'esecuzione del job stesso (Figura 12). Un evolving job avvia un cambiamento nell'allocazione delle risorse richiedendo dinamicamente nodi aggiuntivi al sistema batch e in base a questa richiesta, il sistema batch espande il job. Un job può evolvere per vari motivi. Ad esempio, l'applicazione può aver sostenuto calcoli aggiuntivi come risultato di risultati intermedi che richiedono risorse aggiuntive per poter terminare il job entro il limite di walltime specificato. In alcuni casi, le applicazioni potrebbero non essere in grado di continuare l'esecuzione senza risorse aggiuntive (ad esempio, quando l'applicazione raggiunge i limiti hardware come la memoria). In questi casi, l'applicazione richiede risorse aggiuntive per distribuire i dati e i calcoli e continuare l'esecuzione.

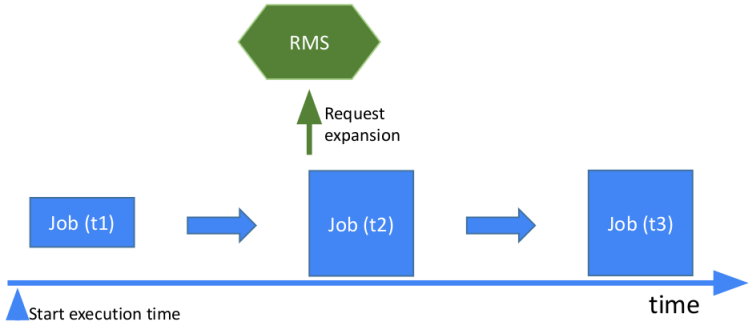

Figura 12: Evolving Job.

Malleable job. I job malleabili sono i job più facili da programmare. Il sistema batch può espandere o ridurre un job malleabile in qualsiasi momento e l'applicazione si adatterà al set di risorse modificato (Figura 13). Il modello di programmazione più comune che consente la malleabilità è OpenMP e uno

futuro è FlexMPI. Quando un numero maggiore di core di un nodo viene messo a disposizione di un processo, il parallelismo può essere modificato in modo trasparente tra le sezioni parallele del programma. Esistono vari modi per specificare i requisiti delle risorse per un job malleabile. Il metodo più comune consiste nello specificare un numero minimo, ideale e massimo di nodi richiesti dal job, in modo che l'allocazione del job possa essere espansa o ridotta all'interno dell'intervallo specificato. Il sistema batch può modificare in modo flessibile l'insieme di risorse di un job malleabile per ottenere un buon utilizzo del sistema e un buon throughput. Anche l'utente ne beneficia in vari modi. Ad esempio, un job malleabile può essere avviato non appena è pronto il numero minimo di nodi necessari ed essere espanso in seguito, man mano che si rendono disponibili altri nodi.

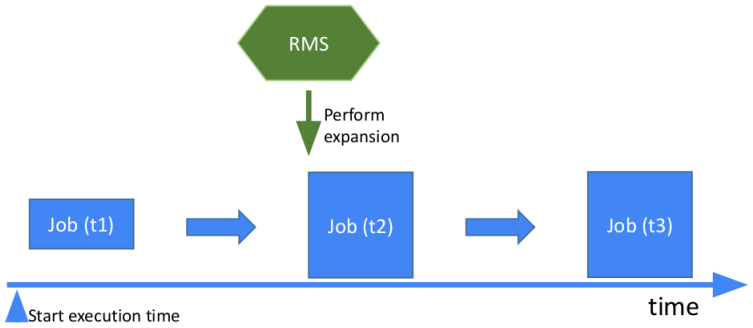

Figura 13: Malleable Job.

Adaptive job. E' un job che viene eseguito con le risorse disponibili, adattandosi di conseguenza. Qualora dovesse essere necessario interrompere il job per motivi di scheduling, viene tenuto in conto delle risorse utilizzate in modo tale che al restart richieda le stesse risorse usate al momento dell'arresto (Figura 14).

Poiché l'allocazione delle risorse per i rigid e moldable job viene effettuata prima dell'inizio del job stesso e non può essere modificata in seguito, viene definita *allocazione statica*. Il processo di espansione o riduzione dell'allocazione delle risorse di un evolving e malleable job (chiamati insieme job adattivi) è definito *allocazione dinamica*.

La Figura 15 riassume quanto detto.

3.1 FlexMPI

FlexMPI è un'estensione MPI, sviluppata nel 2012 e rilasciata per la prima volta nel 2015, che monitora le prestazioni di un'applicazione e utilizza queste informazioni per prendere decisioni in merito alla distribuzione del carico di lavoro e dei dati tra i processi.

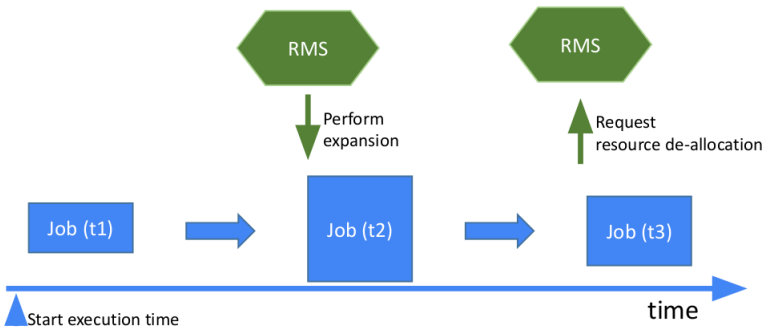

Figura 14: Adaptive Job.

Type	Number of resources	Reconfigurable during runtime	Accepts reconfiguration requests	Can request reconfigurations
Rigid	Fixed	No	-	-
Moldable	Variable	No	-	-
Malleable	Variable	Yes	Yes	No
Evolving	Variable	Yes	No	Yes
Adaptive	Variable	Yes	Yes	Yes

Figura 15: Categorie di job e caratteristiche.

E' usato con applicazioni parallele, basate sul paradigma SPMD, **obbligatoriamente iterative** e che alternano fasi di computazione e fasi di comunicazione. Esempi di tali applicazioni sono Jacobi, Gradiente Coniugato e moto Lagrangiano.

Consiste in: una libreria multithread eseguita all'interno di ogni applicazione MPI; un controller esterno che coordina l'esecuzione di più applicazioni; un'interfaccia con applicazioni di terze parti (scheduler, intelligent controller di ADMIRE).

Caratteristiche di FlexMPI:

- fornisce funzionalità malleabili alle applicazioni MPI;
- fornisce capacità adattive alle applicazioni MPI con politiche diverse: malleabilità rigorosa, high performance, adaptive policies tra cui efficienza, costo e energia, azioni definite dall'utente;
- bilanciamento del carico per piattaforme dedicate e non dedicate;
- monitoraggio dell'applicazione in tempo reale.

L'uso della malleabilità nei propri programmi paralleli (e iterativi nel caso FlexMPI) in due casi:

1. cluster HPC dedicati *solo* ad applicazioni malleabili: se così non fosse, avrei job che allocano risorse incondizionatamente e job malleabili che deallocano e tentano di riallocare risorse che potrebbero non essere disponibili.

2. cluster HPC (anche cloud) con risorse computazionali finite e on-demand: la malleabilità permette di ottimizzare le prestazioni e risparmiare sui costi.

Caratterisitche del Controller:

- application scheduling;
- I/O scheduling.

La malleabilità può essere anche usata per lo scheduling I/O: è possibile per esempio collegare FlexMPI a Hercules[7], si valutano le dimensioni del problema ed è possibile aggiungere o diminuire i nodi del file system.

Il control flow di FlexMPI è mostrato in Figura 16.

Come già detto FlexMPI è usato con applicazioni SPMD iterative e parallele che usano MPI. L'aumento o la diminuzione dei nodi può basarsi su due politiche: riconfigurazione performance-oriented o triggered.

I dati delle applicazioni sono organizzati in strutture dati che vengono registrate in FlexMPI (vettori densi e matrici dense e/o sparse) che ridistribuisce automaticamente a seconda di due condizioni: quando applicato un load balancing o quando i processi vengono creati o rimossi.

Il load balancing è effettuato quando: nuovi processi sono creato o rimossi; le applicazioni sono eseguite su piattaforme con nodi di calcolo a prestazioni diverse; le applicazioni vengono eseguite su una piattaforma con nodi di calcolo non dedicati.

In FlexMPI inoltre le comunicazioni tra i processi sono effettuate con funzioni che "wrappano" le MPI communication call (Figura 17).

Intelligent controller (IC). Esso comunica con gli altri componenti tramite le RCP (Remote Call Procedure) e coopera con il monitoring manager, le applicazioni, gli utenti, il system job scheduler (slurm), l'I/O scheduler, il malleability manager e lo storage. Il suo obiettivo principale è la creazione di una infrastruttura di controllo distribuita fornendo un'unica immagine del sistema attraverso un protocollo di coerenza distribuito ed essendo totalmente trasparente agli utenti. Naturalmente si fa carico del compito di realizzare un efficiente uso delle risorse nei sistemi a larga scala con worlkload dinamici. E' quindi l'intelligent controller a inviare i comandi malleabili e l'application manager ad inviare i dati alla regione malleabile.

[7] Un in-memory file system allocato in RAM da utente (non root).

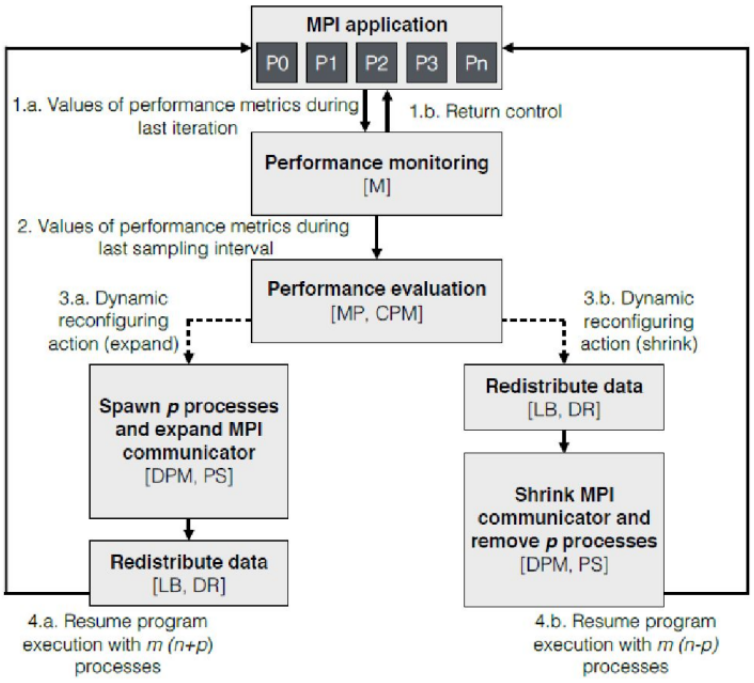

Figura 16: Control flow di FlexMPI.

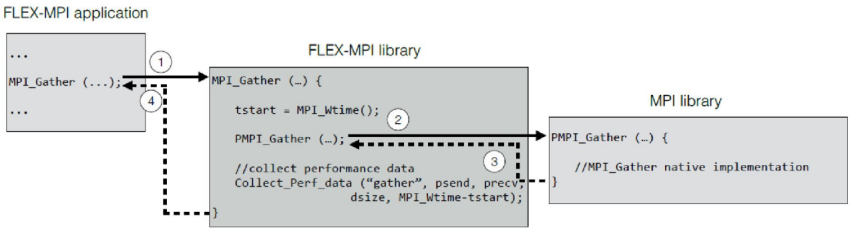

Figura 17: FlexMPI communication call.

La libreria RCP fornisce invece canali di comunicazione tra ogni componente e l'IC. I protocolli per la comunicazione sono due: mono-direzionale e bidirezionale.

Nel primo:

1. l'IC registra RCP generali;
2. il componente/applicazione chiama la dovuta funzione di invio RCP;
3. l'IC memorizza l'indirizzo e l'ID del componente;
4. si ripete l'operazione di invio tante volte quanto necessario;

5. ogni volta l'IC fa il check sulle informazioni da inviare.

Nel secondo:

1. l'IC crea i servizi RCP;
2. il componente/applicazione registra l'RCP e invia il suo indirizzo;
3. l'IC memorizza l'ID del componente e l'indirizzo;
4. viene eseguita l'applicazione da parte del componente;
5. l'IC valuta le nuove azioni da compiere e invia i comandi al componente;
6. il componente riceve i comandi e esegue le dovute azioni.

4 FPP: Ad Hoc File Systems

I backend di storage dei cluster di calcolo parallelo sono ancora basati principalmente su dischi magnetici. Le tecnologie di archiviazione più recenti e più veloci, come gli SSD basati su flash o le non-volatile random access memory (NVRAM), sono distribuite all'interno dei nodi di calcolo. L'inclusione di queste nuove tecnologie di archiviazione nei flussi di lavoro scientifici è purtroppo oggi un compito prevalentemente manuale e la maggior parte degli scienziati non sfrutta quindi i supporti di archiviazione più veloci. Un approccio per includere sistematicamente gli SSD o le NVRAM a livello di nodo nei flussi di lavoro scientifici consiste nel distribuire file system ad hoc su un insieme di nodi di calcolo, che fungono da sistemi di archiviazione temporanea per singole applicazioni o campagne di lunga durata.

4.1 FUSE: File System in User Space

File system in User Space (FUSE) è un'interfaccia software per i sistemi operativi Unix e Unix-like che consente agli utenti non privilegiati di creare i propri file system senza modificare il codice del kernel. Ciò si ottiene eseguendo il codice del file system nello spazio utente, mentre il modulo FUSE fornisce solo un ponte per le interfacce del kernel. FUSE è disponibile per Linux, FreeBSD, OpenBSD, NetBSD (come puff), OpenSolaris, Minix 3, macOS e Windows.

Il FUSE system faceva originariamente parte di AVFS[8] (A Virtual Filesystem). FUSE è stato inserito nel mainstream kernel tree nella versione 2.6.14 (ora 5.14.0) del kernel. Poiché il protocollo kernel-userspace di FUSE è pubblico e soggetto a versioni, un programmatore può scegliere di utilizzare un altro pezzo di codice al posto di libfuse e continuare a comunicare con le strutture FUSE del kernel e "libfuse" fornisce un'interfaccia portabile di alto livello che può essere implementata su un sistema privo di strutture FUSE.

4.1.1 FUSE: Funzionamento

Per implementare un nuovo file system, è necessario scrivere un programma di gestione collegato alla libreria libfuse fornita. Lo scopo principale di questo programma è quello di specificare come il file system deve rispondere alle richieste di lettura/scrittura/stat. Il programma viene utilizzato anche per montare il nuovo file system e, al momento del montaggio del file system, il gestore viene registrato nel kernel. Se ora un utente invia richieste di lettura/scrittura/stat per questo file system appena montato, il kernel inoltra queste richieste di IO al gestore e poi invia la risposta del gestore all'utente.

La Figura 18 mostra un esempio di flusso di lavoro. La richiesta dello spazio utente di elencare i file (ls -l /tmp/fuse) viene reindirizzata dal kernel attraverso il VFS a FUSE. FUSE esegue quindi il programma handler registrato (./hello) e gli passa la richiesta (ls -l /tmp/fuse). Il programma handler restituisce a

[8] AVFS è un sistema che consente a tutti i programmi di guardare all'interno di file archiviati o compressi o di accedere a file remoti senza ricompilare i programmi o modificare il kernel.

FUSE una risposta che viene poi reindirizzata al programma userspace che ha originariamente effettuato la richiesta.

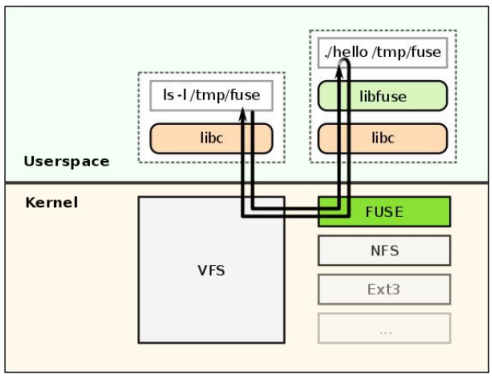

Figura 18: Funzionamento di FUSE.

FUSE è quindi particolarmente utile per scrivere filesystem virtuali. A differenza dei filesystem tradizionali che si preoccupano principalmente di organizzare e memorizzare i dati su disco, i filesystem virtuali non memorizzano realmente i dati per conto proprio. Agiscono infatti come un tramite fra l'utente ed il filesystem reale sottostante. FUSE è un sistema molto potente: virtualmente ogni risorsa disponibile ad essere implementata sfruttando FUSE può divenire un filesystem virtuale.

4.1.2 FUSE: Applicazioni

On-disk file systems. I file system convenzionali su disco possono essere implementati nello spazio utente con FUSE, ad esempio per motivi di compatibilità o di licenza.

- **Linear Tape File System**: Consente di accedere ai file memorizzati su nastro magnetico in modo simile a quelli su disco o su unità flash rimovibili.
- **NTFS-3G e Captive NTFS**: consentono l'accesso ai filesystem NTFS.
- **retro-fuse**: è un filesystem user-space che fornisce un modo per montare filesystem creati da antichi sistemi Unix su sistemi operativi moderni. La versione attuale di retro-fuse supporta il montaggio di filesystem creati dalla quinta, sesta e settima edizione di Research Unix di BTL, nonché da sistemi basati su 2.9BSD e 2.11BSD.

FUSE permette di definire file system ad-hoc in spazio utente che permettono di usare un qualsiasi tipo di file system di back-end, in modo trasparente all'utente. Quindi in questo modo è possibile accedere ai dati come per un file system classico, ma poi il modo con cui i dati sono memorizzati effettivamente nel backend è gestito in modo personalizzato.

FUSE può essere anche usato quando si vuole un file system che non memorizzi i dati in modo permanente sul sistema fisico.

FUSE può essere utilizzato per client di file system remoti/distribuiti, ma anche per la creazione di file system ad-hoc per l'HPC. Degli esempi sono: *BeeOND*, *GekkoFS* e *BurstFS*. Quindi, il futuro di FUSE è associato alla distribuzione remota di dati su sitemi di calcolo ad alte prestazioni.

Archive and backup file systems. I filesystem FUSE, inoltre, possono essere usati per esporre il contenuto di archivi o set di backup senza dover prima estrarre i file. Degli esempi sono: archivemount; Atlas (Rubrik backup software); Borg (backup software); Restic; SPFS (file system for Spectrum Protect).

4.2 DAGonFS

DAGonStar è un altro workflow engine: un software che gestisce e monitora lo stato delle attività in un flusso di lavoro e determina a quale nuova attività passare in base ai processi definiti, ossia un orchestratore di workflow. DAGonFS (Figura 14) è un file system ad-hoc per workflow engine che sfrutta la libreria FUSE per creare un file system distribuito in RAM e MPI per garantire la scalabilità dello storage e delle prestazioni.

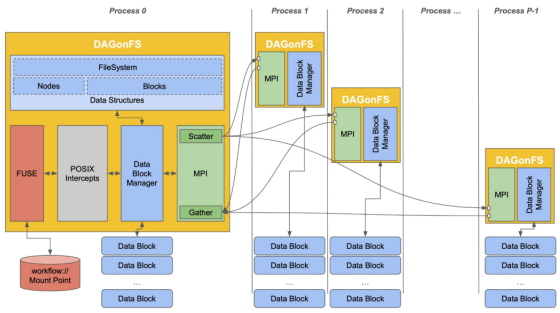

Figura 19: Architettura di DAGonFS.

4.3 ADMIRE

4.3.1 ADMIRE: Motivazioni

La crescente necessità di elaborare insiemi di dati estremamente grandi è oggi uno dei principali fattori che spingono a costruire sistemi HPC exascale. Tuttavia, le gerarchie di archiviazione piatte presenti nelle architetture HPC classiche non soddisfano più i requisiti di prestazione delle applicazioni di elaborazione dati. L'accesso non coordinato ai file e la larghezza di banda limitata rendono il file system parallelo centralizzato di back-end un serio collo di bottiglia. Allo stesso tempo, le emergenti gerarchie di storage multi-tier hanno il potenziale per

rimuovere questa barriera. Ma la massimizzazione delle prestazioni richiede ancora un controllo accurato per evitare la congestione e bilanciare le prestazioni di calcolo e di archiviazione. Purtroppo, mancano ancora interfacce e politiche adeguate per la gestione di uno stack I/O potenziato.

Alcune soluzioni potrebbero essere:

- Alluxio è una soluzione di archiviazione che si colloca tra i framework di calcolo e i data store persistenti e che mira a ridurre la complessità delle API di archiviazione sfruttando la velocità della memoria.
- Hermes si concentra sull'implementazione di un sistema di archiviazione basato su MRAM che migliora le prestazioni del file system attraverso l'uso efficace dei dispositivi MRAM.
- WekaIO fornisce un'architettura di archiviazione ad alte prestazioni.

Tutti però mancano di meccanismi di località dei dati e delle caratteristiche dei file system ad-hoc.

La malleabilità può: adattare il sistema di storage alle esigenze di I/O dell'applicazione durante le diverse fasi di esecuzione; estendere o ridurre il file system ad hoc in fase di esecuzione, seguendo le richieste di I/O dell'applicazione; accelerare le fasi ad alta intensità di I/O; ridurre l'ingombro in memoria del backend di archiviazione in-memory.

4.3.2 ADMIRE: Abilita le Azioni di Sistema Basate sui Dati

L'obiettivo principale del progetto ADMIRE è stabilire il controllo per la massimizzazione delle prestazioni creando uno stack I/O attivo che regoli dinamicamente i requisiti di calcolo e di archiviazione attraverso un coordinamento globale intelligente, la malleabilità di calcolo e di I/O e la programmazione delle risorse di archiviazione lungo tutti i livelli della gerarchia di archiviazione. Per raggiungere questo obiettivo, è stato sviluppato un software-defined framework (Figura 20) basato sui principi del monitoraggio e del controllo scalabili, sulla separazione dei percorsi di controllo e di dati e sull'orchestrazione dei componenti e delle applicazioni chiave del sistema attraverso punti di controllo incorporati.

4.3.3 ADMIRE: Ad-Hoc Storage Systems

Per quanto concerne lo storage, sono stati implementati storage system ad-hoc temporanei con caratteristiche di resilienza e che supportano tecnologie di archiviazione locale veloce come RAM, NVM, SSD e altre.

Nel contesto ADMIRE si differenziano quattro sistemi storage: GekkoFs, Hercules IMSS, Expand e dataClay. I primi tre sono sono ad-hoc file system, mentre l'ultimo è un object-oriented data store.

La caratteristica principale è che tutti questi sistemi storage ad-hoc supportano le stesse API (POSIX). E' comunque possibile avere interfacce I/O custom, ma poi potrebbero essere necessari dei cambiamenti nell'applicazione.

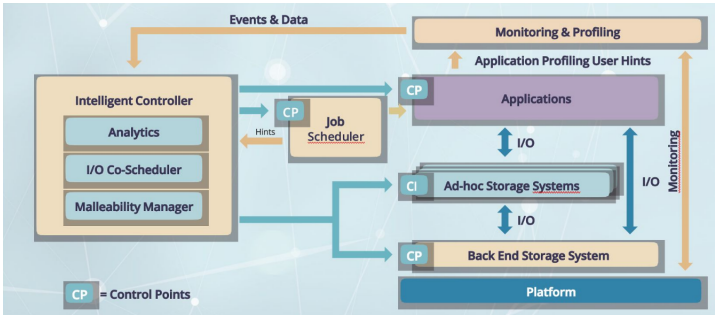

Figura 20: Framework ADMIRE.

GekkoFS. E' un file system distribuito a livello utente altamente scalabile per cluster HPC. GekkoFS è in grado di aggregare la capacità di I/O locale e le prestazioni dei nodi di calcolo per produrre uno spazio di archiviazione ad alte prestazioni per le applicazioni. Con GekkoFS, le applicazioni e le simulazioni HPC possono essere eseguite in isolamento l'una dall'altra per quanto riguarda l'I/O, riducendo le interferenze e migliorando le prestazioni.

A differenza dei file system paralleli generici, un file system GekkoFS è di natura effimera (quindi temporaneo). In altre parole, la durata di un'istanza del file system GekkoFS è legata alla durata dell'esecuzione dei suoi processi server GekkoFS, che vengono tipicamente generati all'avvio di un lavoro HPC e chiusi al suo termine. Ciò significa che gli utenti devono copiare tutti i file che devono essere conservati oltre la durata del lavoro da GekkoFS a un file system permanente.

L'**architettura** di GekkoFS è mostrata in Figura 21. E' costituito da due componenti principali: una libreria client e un processo server, chiamato *daemon*. GekkoFS viene eseguito interamente nello spazio utente e può quindi essere utilizzato da qualsiasi utente senza bisogno di supporto amministrativo. Di conseguenza, un'applicazione deve utilizzare GekkoFS tramite la libreria di interposizione client attraverso la variabile d'ambiente LD_PRELOAD. La libreria client intercetta quindi tutte le chiamate di I/O e controlla se si trovano all'interno del punto di montaggio virtuale. In caso contrario, le chiamate di I/O vengono inoltrate al sistema operativo sottostante.

dataClay. È un archivio di oggetti distribuito ed è stato progettato per nascondere i dettagli della distribuzione e sfruttare l'infrastruttura sottostante, sia essa un cluster HPC o un ambiente altamente distribuito come l'edge-to-cloud.

Gli oggetti in dataClay sono arricchiti di semantica, compresa la possibilità di allegare ad essi codice utente arbitrario e quindi tali oggetti sono la combinazione di dati e metodi. Questo permette di ridurre i costi di comunicazione e migliora le performance dell'intera applicazione. Permette di gestire gli oggetti

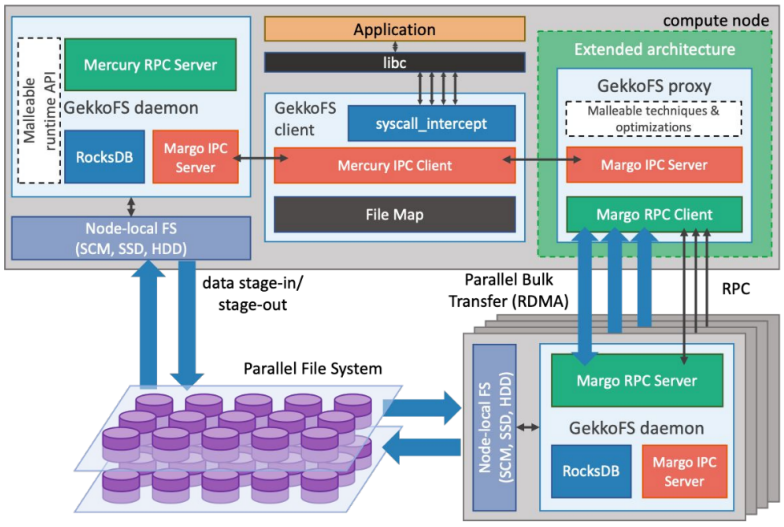

Figura 21: Architettura di GekkoFS.

direttamente in memoria durante l'esecuzione e questo evita le trasformazioni sui dati e riduce il numero di accessi al disco.

DataClay è implementato a livello utente, quindi è visibile alle applicazioni che utilizzano la sua libreria client. Può essere distribuito in due modi diversi: come servizio o come sistema di archiviazione effimero.

L'**architettura** di dataClay è mostrata in Figura 22. I componenti principali di dataClay sono il *Logic Module* e i *Backend*.

Il Logic Module è l'autorità centrale per i metadati. Contiene informazioni sulla posizione degli oggetti e sui loro identificatori (*Metadata Manager*), che sono interni, unici e generati automaticamente dal sistema. Inoltre gestisce anche le informazioni semantiche sugli oggetti, cioè la loro struttura e il codice utente associato (*Class Manager*). Infine il Logic Module è anche responsabile della gestione delle sessioni utente (*Session Manager*).

Ogni Backend è composto da una *Storage Location*, dove vengono memorizzati gli oggetti, e da un *Execution Environment*, che gestisce le richieste di esecuzione sul sottoinsieme di oggetti gestiti dal backend.

Hercules. E' un acceleratore I/O in-memory per i dati volatili e fornisce un supporto POSIX-based per la portabilità. Viene usato per gestire i workflow HPC, come checkpoint engine, per il locking distribuito.

Le caratteristiche principali sono:

- supporta l'attach/detach dei server di storage;
- supporta diverse policies per la distribuzione dei dati;

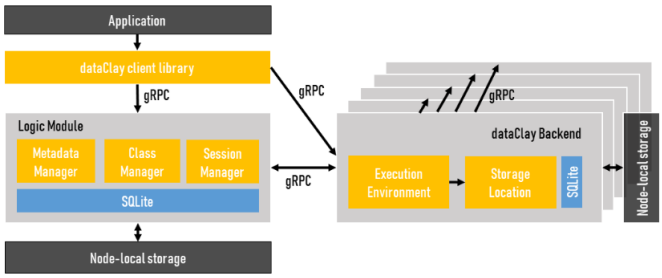

Figura 22: Architettura di dataClay.

- storage backend efficiente basato su GLIB;
- integrato con SLURM;
- replicazione dei dati.

L'**architettura** di Hercules è mostrata in Figura 23 ed è composto da due architectural layer: Client layer e Back-end layer.

- *Client layer*. Le applicazioni client gestiscono istanze IMSS e dataset attraverso una libreria client IMSS. L'API fornisce una serie di operazioni per creare, unire, ottenere, impostare e rilasciare dati, dataset e istanze IMSS. In ogni sessione, i client creano e uniscono più istanze IMSS. Un'istanza IMSS è definita come un'entità di archiviazione dedicata effimera caratterizzata da più server distribuiti lungo un insieme di macchine definite dall'utente che utilizzano la memoria principale per memorizzare i set di dati. Un'istanza IMSS è identificata da un URI (Uniform Resource Identifier) unico.
- *Back-end layer*. Ogni istanza IMSS è caratterizzata da più server di archiviazione IMSS. Ognuno di essi memorizza più data block di diversi dataset. Ogni server IMSS distribuisce un thread dispatcher che distribuisce e bilancia le richieste di connessione dei client tra i thread worker seguendo una politica di round-robin. Inoltre, i thread worker appartenenti allo stesso server associano gli identificatori dei blocchi di dati alle posizioni di memoria in un map-based memory container.

Hercules, dopo un po', è migrato dal **modello di comunicazione** ZeroMQ a UCX.

Unified Communication X (UCX) è un framework di comunicazione ottimizzato e collaudato per le reti moderne, con larghezza di banda elevata e bassa latenza. Espone una serie di primitive di comunicazione astratte che utilizzano il meglio delle risorse hardware e degli offload disponibili. Queste includono RDMA (InfiniBand e RoCE), TCP, GPU, memoria condivisa e operazioni atomiche di rete. UCX facilita lo sviluppo rapido fornendo un'API di alto livello,

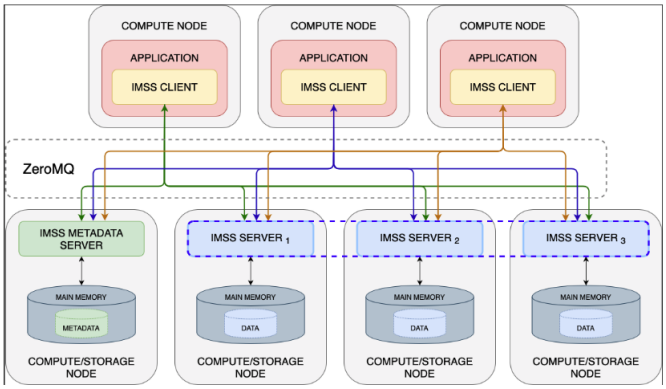

Figura 23: Architettura di Hercules.

mascherando i dettagli di basso livello, pur mantenendo alte le prestazioni e la scalabilità.

La Figura 24 mostra la nuova **architettura**. I componenti principali del livello di comunicazione sono i worker UCX. Un worker UCX astrae un'istanza di risorse di rete, come un host, un'interfaccia di rete o più risorse, come più interfacce di rete.

Il frontend layer si appoggia a due worker UCX indipendenti per consentire la comunicazione punto-punto con i server di dati e metadati. Questo meccanismo garantisce una trasmissione isolata utilizzando code di comunicazione diverse. All'inizializzazione, la libreria client richiede al thread del dispatcher backend l'indirizzo del worker UCX. Questo indirizzo viene utilizzato per la creazione di endpoint su entrambi i lati che rappresentano una connessione da un worker locale a un worker remoto.

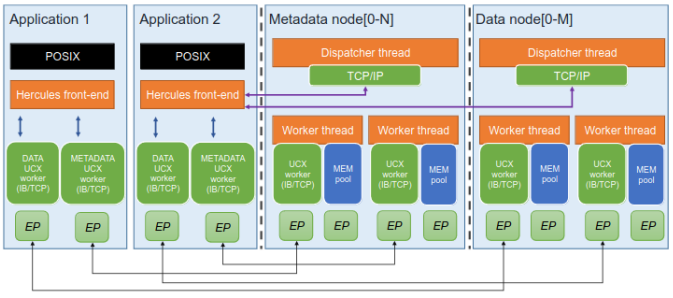

Figura 24: Architettura di Hercules con UCX.

I benefici nell'uso di UCX in Hercules sono i seguenti:

- Disponibilità di più interfacce/protocolli di rete (sono supportati TCP/IP,

Omnipath e Infiniband).

- Trasferimenti di messaggi a copia zero di pacchetti di dati di grandi dimensioni (maggiore o uguale di 1 Mb).
- Eliminazione delle copie interne dall'applicazione al livello di rete.
- Comunicazione asincrona tra peer.
- Isolamento RDMA QoS.
- Comunicazione end-point/two-sided-based.

In Hercules sono state inserite alcune **tecniche di malleabilità** per facilitare la modifica dinamica del numero di nodi di dati in fase di esecuzione. In base alle esigenze di I/O dell'applicazione, Hercules può *espandere o ridurre* il numero di nodi di dati per aumentare o ridurre il throughput di I/O dell'applicazione.

Le **politiche di distribuzione del set di dati** incluse in Hercules IMSS sono:

- ROUND ROBIN: i blocchi di dati sono distribuiti tra i server Hercules.
- BUCKETS: ogni set di dati è suddiviso in un numero di chunk pari al numero di server. Ogni chunk è composto da un numero consecutivo di blocchi di dati, equamente distribuiti, e quindi ogni chunk viene assegnato a un unico server.
- HASHED: su ogni blocco di dati viene applicata un'operazione di hash per scoprire il server mappato.
- CRC16bits e CRC64bits: simile alla politica HASHED, ma viene applicata un'operazione CRC a sedici/sessantaquattro bit sulla data block key.
- LOCALE: ogni blocco di dati viene gestito dal server Hercules in esecuzione nello stesso nodo del client.

Hercules IMSS offre due **strategie di distribuzione** per adattare il sistema di storage ai requisiti dell'applicazione: application-detached e application-attacched.

Con la strategia *application-detached*, i client IMSS vengono distribuiti sugli stessi nodi di calcolo dell'applicazione, utilizzandoli per sfruttare tutte le risorse di calcolo disponibili all'interno di un cluster HPC, mentre i server IMSS si occuperanno di memorizzare i dataset dell'applicazione e di abilitare l'esecuzione dello storage nei nodi offshore dell'applicazione.

Con la strategia *application-attached*, invece, ogni nodo applicativo include sia un client IMSS e sia un server IMSS, distribuiti come thread all'interno dell'applicazione.

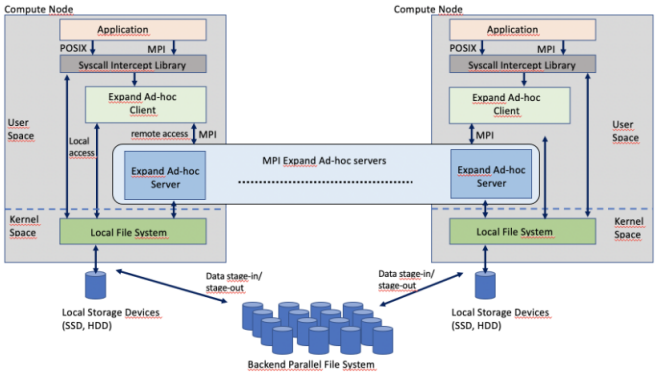

Figura 25: Architettura di Expand.

Expand. E' un file system ad-hoc parallelo e distribuito basato su server storage standard. La struttura si basa su una serie di server di dati in esecuzione sui nodi di calcolo che comunicano tra loro utilizzando MPI.

La Figura 25 ne mostra l'**architettura**. Le partizioni di dati parallele vengono create sui server ad hoc su dispositivi di archiviazione locali (HDD o SSD) utilizzando i servizi forniti dal sistema operativo locale. Expand combina quindi diversi server MPI ad hoc per fornire una partizione parallela generica. Ogni server fornisce una o più directory che vengono combinate per costruire una partizione distribuita da utilizzare nei nodi di calcolo. Tutti i file del sistema sono distribuiti a strisce su tutti i server ad hoc per facilitare l'accesso parallelo, e ogni server memorizza concettualmente un sottofile del file parallelo. Un file è composto da diversi sottofile, uno per ogni server ad hoc. Tutti i sottofile sono completamente trasparenti per gli utenti di Expand. Su una partizione parallela, l'utente può creare dei file strippati con layout ciclico. In questi file, i blocchi sono distribuiti nella partizione secondo uno schema round-robin.