**FCAPP - Future Programming Paradigms:** **Parallel Programming Models**

**01-FCAPP-Parallel\_Programming\_Models**

**1. Control**

**🔹 *Come viene creata la parallelizzazione?***

* **Implicita (hardwired)**: la parallelizzazione è gestita automaticamente, ad esempio da un compilatore ottimizzante o da un sistema runtime. L’utente scrive codice “normale” e il sistema lo esegue in parallelo.
  + Esempio: compilatori che sfruttano automaticamente il vectorization.
* **Esplicita**: il programmatore gestisce direttamente la creazione di thread, sincronizzazione e divisione dei compiti.
  + Esempio: usare OpenMP per creare thread o MPI per distribuire dati tra processi.

**🔹 *Quali ordinamenti esistono tra le operazioni?***

* In un ambiente parallelo, l’**ordine di esecuzione delle istruzioni può variare** tra i thread, a meno che non venga imposto esplicitamente.
* Questo crea problemi come **race conditions** se due thread accedono a dati condivisi senza sincronizzazione.

N.B: Una **race condition** si verifica quando **due o più thread/processi accedono contemporaneamente a una risorsa condivisa**, e **l’esito del programma dipende dall’ordine non deterministico in cui questi accessi avvengono**.

**🔹 *Come si sincronizzano i diversi thread di controllo?***

* Tramite **primitive di sincronizzazione**:
  + **Lock/mutex**: per garantire accesso esclusivo a sezioni critiche.
  + **Semafori**: per gestire risorse con accesso limitato.
  + **Barriere**: per sincronizzare più thread a un certo punto del programma.
  + **Condition variables, atomic operations**, ecc.

**🔐 2. Naming Data**

**🔹 *Quali dati sono privati e quali condivisi?***

* **Privati**: ogni thread/processo ha una sua copia. Esempi: variabili locali in una funzione.
* **Condivisi**: visibili e accessibili da più thread. Esempi: variabili globali, heap condiviso.

**🔹 *Come viene gestito l’accesso ai dati condivisi?***

* Tramite **indirizzi comuni** (in memoria condivisa) o **comunicazione esplicita** (in sistemi a memoria distribuita).
* È cruciale evitare accessi concorrenti non sincronizzati ai dati condivisi.

**⚙️ 3. Operations on Data**

**🔹 *Quali sono le operazioni di base sui dati condivisi?***

* **Lettura e scrittura**: le operazioni più semplici ma che possono creare problemi se fatte simultaneamente da più thread.
* **Update/incrementi**: ad esempio, x = x + 1 sembra semplice, ma in realtà include lettura, somma e scrittura – quindi **non è atomica**.

**🔹 *Quali operazioni sono considerate atomiche?***

* Un’**operazione atomica** è un’operazione che avviene tutta in una volta, senza interferenze.
  + Esempi: fetch\_and\_add, compare\_and\_swap.
* Le operazioni atomiche sono fondamentali per **evitare race conditions** senza usare lock complessi.

**💰 4. Cost**

**🔹 *Come valutiamo il costo della parallelizzazione?***

* **Tempo umano (man hours)**: scrivere e debuggare codice parallelo è più complesso.
* **Costo software**: strumenti, librerie, debugger specifici.
* **Costo hardware**: CPU multi-core, GPU, interconnessioni di rete nei cluster.
* **Overhead di comunicazione e sincronizzazione**: troppa sincronizzazione può annullare i benefici della parallelizzazione.
* **Scalabilità**: quanto bene cresce la prestazione all’aumentare del numero di core o nodi.

**✅ In sintesi:**

La programmazione parallela non è solo “lanciare tanti thread”. Richiede attenzione a **controllo, visibilità dei dati, sincronizzazione, atomicità e costo delle scelte fatte**. Ogni errore in queste aree può portare a bug difficili o prestazioni deludenti.

**🧰 Programming Model: Definizione**

Un **modello di programmazione** è un’**astrazione dell’hardware** usata dal programmatore per scrivere codice parallelo in modo portabile. Non descrive come è fatto il computer, ma **come viene percepito dal programmatore**.

**1. Multiprogramming Model**

📌 **Caratteristiche:**

* Si basa su **compiti indipendenti** che girano in parallelo (o meglio, *concorrenza*, non vera *cooperazione*).
* Non c’è **comunicazione o sincronizzazione** tra i task a livello di programma.
* È il sistema operativo a gestire i processi in modo concorrente.

🧾 **Esempio pratico:**

* Un web server gestisce tante richieste da browser: ogni richiesta è servita da un processo o thread indipendente, che non ha bisogno di coordinarsi con gli altri.

🧠 **Nota:** Non è un vero modello di *programmazione parallela cooperativa*, ma è importante da capire come base.

**2. Shared Address Space (Shared Memory) Programming**

📌 **Caratteristiche:**

* I task (tipicamente **thread**) accedono a **un’unica memoria condivisa**.
* La comunicazione avviene leggendo/scrivendo **variabili comuni**.
* Serve **sincronizzazione esplicita** per evitare conflitti (es. race condition).

📋 **Metafora utile:** è come una **bacheca** in un ufficio: ognuno scrive/legge messaggi, ma bisogna organizzarsi per evitare sovrapposizioni o confusione.

**3. Message Passing Programming**

📌 **Caratteristiche:**

* Ogni processo ha **memoria privata** e comunica tramite **messaggi espliciti**.
* Nessuna memoria condivisa.
* La comunicazione può essere:
  + **Ordinata (connection-oriented)** → come una telefonata: entrambe le parti devono essere attive.
  + **Non ordinata (connectionless)** → come un’email o una cassetta postale: il messaggio arriva anche se il destinatario non è “online”.

📋 **Metafora utile:** è come inviare **lettere o telefonate** tra uffici diversi.

🧾 **Esempi:**

* **MPI (Message Passing Interface)**: standard molto usato nei supercomputer.
* **PVM (Parallel Virtual Machine)**: un po’ più vecchio, simile a MPI.

⚠️ **Sfide:**

* Gestione esplicita della comunicazione
* Possibilità di **deadlock**
* Maggiore complessità se il numero di processi è alto

Immagine che contiene testo, schermata, Carattere, linea

Il contenuto generato dall'IA potrebbe non essere corretto.

**🧩 Flynn's Taxonomy**

Michael Flynn (1966) ha proposto una classificazione dei computer in base a **quante istruzioni** e **quanti dati** vengono processati contemporaneamente:

Immagine che contiene testo, schermata, Carattere, linea

Il contenuto generato dall'IA potrebbe non essere corretto.

**1. SISD – *Single Instruction Single Data***

Questo è il modello più semplice e tradizionale: una **sola CPU** esegue un **flusso di istruzioni** su **un solo dato alla volta**. È l’architettura tipica dei **computer sequenziali**, come la classica **macchina di Von Neumann**, e dei personal computer commerciali **fino a circa il 2010**.

In questa categoria **non c’è parallelismo** reale. Anche se alcuni sistemi possono avere più core o usare tecniche come il pipelining, se l’esecuzione rimane lineare e sequenziale, il modello rientra ancora in SISD.

**2. SIMD – *Single Instruction Multiple Data***

In questa architettura, un **unico flusso di istruzioni** viene applicato **simultaneamente a più dati**. In pratica, la stessa operazione viene eseguita in parallelo su insiemi diversi di dati, rendendola molto adatta per operazioni su array o vettori, come nel caso del calcolo scientifico o del trattamento multimediale.

Sono esempi di questa architettura gli **array processor**, i **supercomputer vettoriali** e le estensioni SIMD presenti nelle CPU moderne, come **SSE**, **MMX** o **AVX**.

**3. MISD – *Multiple Instruction Single Data***

Questa categoria è per lo più **teorica** e **non trova applicazioni reali significative**. L’idea è che istruzioni diverse vengano eseguite contemporaneamente **sullo stesso dato**. Alcuni sistemi fault-tolerant potrebbero rientrare in questa definizione, ma in pratica **non esistono esempi concreti** di largo impiego.

**4. MIMD – *Multiple Instruction Multiple Data***

Questa è l’architettura più **flessibile e diffusa nei sistemi moderni**. In un sistema MIMD, **ogni processore** può eseguire **istruzioni diverse su dati diversi**, in modo totalmente indipendente. È il caso dei **sistemi multiprocessore**, dei **supercomputer** e dei **cluster HPC**.

Una forma molto comune di programmazione per queste architetture è lo stile **SPMD** (*Single Program Multiple Data*), in cui ogni processore esegue la **stessa applicazione** ma lavora su un **sottoinsieme diverso di dati**. Questo modello è tipico, ad esempio, nei programmi sviluppati con **MPI**.

**Immagine che contiene testo, ricevuta, schermata, Carattere

Il contenuto generato dall'IA potrebbe non essere corretto.**

**🧩 MIMD vs SIMD: due modi diversi di fare parallelismo**

Nel calcolo parallelo, **MIMD** e **SIMD** rappresentano **due approcci complementari** all'elaborazione su più processori. Entrambi sfruttano più unità di calcolo per aumentare le prestazioni, ma lo fanno in modo molto diverso, sia per struttura che per strategia.

**🔧 MIMD (Multiple Instruction Multiple Data) – Parallelismo a task**

Nel modello **MIMD**, ogni processore esegue un **task indipendente**, cioè una **serie di istruzioni diverse su dati diversi**. Questo approccio è alla base del **parallelismo a livello di task (task parallelism)**, in cui un’applicazione viene suddivisa in **sottocompiti differenti**, eseguiti contemporaneamente su più core o thread.

💡 Immaginalo come una squadra in cui ognuno fa un lavoro diverso: un cuoco, un cameriere e un receptionist. Collaborano, ma eseguono attività diverse.

**Caratteristiche principali:**

* Ogni processore può eseguire un programma diverso (o lo stesso con flussi di controllo diversi).
* È tipico nei sistemi **multicore, multiprocessore** e nei **cluster HPC**.
* Può usare sia **shared memory** (es. OpenMP) che **message passing** (es. MPI).
* La **sincronizzazione è esplicita**: il programmatore deve usare meccanismi come lock, semafori o barriere.

📌 Questo modello è alla base della programmazione **thread-level** in ambienti MIMD, spesso organizzata secondo il paradigma **fork-join**: il programma si divide in thread (fork), esegue i compiti in parallelo, poi si riunisce (join) per continuare.

**🔄 SIMD (Single Instruction Multiple Data) – Parallelismo a dati**

Nel modello **SIMD**, **una singola istruzione** viene eseguita **simultaneamente su più dati diversi**. Questo è il cuore del **parallelismo a livello di dati (data parallelism)**, dove il lavoro è distribuito su elementi di un array o una matrice, eseguito da molti processori in parallelo ma in modo uniforme.

💡 Immaginalo come una catena di montaggio in cui tutti fanno la stessa cosa su pezzi diversi: ognuno avvita la stessa vite, ma su oggetti diversi.

**Caratteristiche principali:**

* Tutti i processori eseguono la stessa istruzione, su blocchi di dati differenti.
* È tipico delle **GPU**, delle **istruzioni vettoriali SIMD** (SSE, AVX) e delle **macchine vettoriali**.
* Lavora molto bene su strutture dati regolari come array, matrici, vettori.
* Usa **shared memory o indirizzamento logico condiviso**.
* La **sincronizzazione è implicita**: i thread avanzano insieme, in lock-step.

**🎯 In sintesi**

* **Usa MIMD** quando i compiti da eseguire sono diversi (ad esempio: simulazioni complesse, web server).
* **Usa SIMD** quando il compito è uniforme e i dati sono tanti (ad esempio: elaborazione di immagini o array numerici).

Esempio saltabile.

Immagine che contiene testo, Carattere, linea, schermata

Il contenuto generato dall'IA potrebbe non essere corretto.

Immagine che contiene Carattere, bianco, design, tipografia

Il contenuto generato dall'IA potrebbe non essere corretto.Immagine che contiene disegno, schizzo, linea, diagramma

Il contenuto generato dall'IA potrebbe non essere corretto.

**🧩 Passaggi della decomposizione parallela:**

**1. Divisione del lavoro – Parallel Decomposition**

* Hai **P processori** (o thread).
* Dividi l’array in **blocchi da N/P elementi**.
* Ogni processore calcola la **somma parziale** del suo blocco.

📌 **Esempio con N = 1000, P = 4**

* P0 lavora su a[1..250]
* P1 su a[251..500]
* P2 su a[501..750]
* P3 su a[751..1000]

Immagine che contiene testo, schermata, Carattere, linea

Il contenuto generato dall'IA potrebbe non essere corretto.

**3. Raccolta dei risultati – Reduction**

* Un **processore “master”** (o una riduzione collettiva) somma tutti i Aj per ottenere A.

Immagine che contiene testo, Carattere, linea, schermata

Il contenuto generato dall'IA potrebbe non essere corretto.

**🧵 Multiprogramming Model**

Il **modello di multiprogrammazione** è uno dei più **antichi e diffusi** nella storia dell'informatica. La sua caratteristica principale è la **coesistenza di più programmi (o task)** attivi nello stesso momento su una singola macchina.

Tuttavia, questi programmi sono completamente **indipendenti** tra loro: **non comunicano** e **non si sincronizzano** a livello applicativo. In pratica, la CPU **passa da un programma all’altro** quando quello corrente è momentaneamente bloccato (ad esempio, in attesa di I/O), in modo da **non sprecare risorse computazionali**.

**👥 Multiprogramming nei sistemi paralleli**

Nel contesto del **parallel computing**, la multiprogrammazione si concretizza spesso in una **shared memory task parallel model**, tipico delle architetture **MIMD**.

Qui ogni **processo** può essere suddiviso in **più thread**, che:

* **Condividono la memoria globale**.
* Hanno anche **variabili private** (es. lo stato del thread e variabili locali nella stack).
* Si **coordinano esplicitamente** attraverso operazioni di sincronizzazione.

**🏗️ Architetture per Shared Memory**

**🧠 Differenze tra UMA, NUMA e DSM**

Quando si parla di **sistemi a memoria condivisa**, è fondamentale distinguere tra **come** la memoria è distribuita tra i processori e **come** viene acceduta. Le architetture UMA, NUMA e DSM rappresentano **modelli diversi di organizzazione e accesso alla memoria** in ambienti multiprocessore.

**🟩 UMA – *Uniform Memory Access***

Nel modello **UMA**, tutti i processori **condividono un’unica memoria centrale** e **ci accedono con lo stesso tempo e alla stessa velocità**, indipendentemente da quale processore effettua la richiesta. Questo tipo di architettura è tipico dei sistemi **SMP** (*Symmetric Multiprocessors*), in cui più CPU lavorano fianco a fianco e condividono equamente la memoria principale.

* **Non esiste una memoria privata o locale**: tutte le variabili, anche quelle “locali” a un thread, vengono allocate nella memoria condivisa.
* È **facile da programmare** e molto efficiente per un numero limitato di processori.
* Ma **non scala bene**: man mano che il numero di processori aumenta aumentano i conflitti di accesso alla memoria e il bus si satura.

📌 È l’architettura tipica dei sistemi multicore più semplici, o di server con poche CPU condivise.

Immagine che contiene diagramma, testo, Piano, Disegno tecnico

Il contenuto generato dall'IA potrebbe non essere corretto.

**🟨 NUMA – *Non Uniform Memory Access***

In una macchina **NUMA**, la memoria è **divisa in blocchi locali** associati a ciascun processore o gruppo di processori. Ogni CPU può accedere alla propria memoria **più velocemente** rispetto alla memoria collegata ad altri processori.

* L'accesso alla **memoria locale è più veloce**.
* L’accesso a **memoria remota è possibile**, ma avviene con maggiore latenza.
* È una **soluzione scalabile** per architetture con molti core o socket multipli (tipica nei moderni server e supercomputer).
* La programmazione è più complessa: bisogna tener conto della **località dei dati** per evitare rallentamenti.

📌 I sistemi NUMA moderni spesso includono **meccanismi per "migrare" i dati** nella memoria più vicina al processore che li usa (NUMA-aware scheduling e data placement).

Immagine che contiene diagramma, Piano, Disegno tecnico, linea

Il contenuto generato dall'IA potrebbe non essere corretto.

**🟦 DSM – *Distributed Shared Memory***

Il modello **DSM** è una **via di mezzo tra memoria condivisa e distribuita**: anche se la memoria è fisicamente distribuita tra nodi distinti (ad esempio in un cluster), viene **vista come un unico spazio di memoria condiviso** a livello logico.

* I dati non sono nella stessa RAM, ma in **memorie fisicamente separate** collegate tramite rete.
* L’accesso a dati remoti avviene tramite **meccanismi di comunicazione** (hardware o software).
* Richiede un **protocollo di coerenza** per garantire che le copie di dati condivisi rimangano consistenti.
* Ha il vantaggio di essere **scalabile**, **più economico** di un sistema SMP, e consente di **scrivere programmi portabili** su sistemi diversi.

📌 È usato in ambienti di **calcolo distribuito**, come nei cluster HPC con programmazione parallela basata su librerie come **MPI + DSM software**.

Immagine che contiene diagramma, testo, Piano, linea

Il contenuto generato dall'IA potrebbe non essere corretto.

Immagine che contiene testo, schermata, Carattere, numero

Il contenuto generato dall'IA potrebbe non essere corretto.

Immagine che contiene testo, Carattere, schermata, algebra

Il contenuto generato dall'IA potrebbe non essere corretto.

Immagine che contiene testo, schermata, Carattere, linea

Il contenuto generato dall'IA potrebbe non essere corretto.

Immagine che contiene testo, schermata, Carattere

Il contenuto generato dall'IA potrebbe non essere corretto.

Immagine che contiene testo, schermata, Carattere, numero

Il contenuto generato dall'IA potrebbe non essere corretto. Immagine che contiene testo, schermata, Carattere, numero

Il contenuto generato dall'IA potrebbe non essere corretto.

**⚠️ Problema evidenziato**

* La riga in rosso A := A[1] + A[2] **può essere eseguita da Thread 1 prima che Thread 2 abbia finito di calcolare A[2]**, portando a un risultato errato.
* Questo è un classico esempio di **race condition**.

**✅ Soluzioni possibili**

1. **Sincronizzazione con una barriera**: assicurarsi che entrambi i thread abbiano terminato prima di sommare A[1] + A[2].
2. **Uso di operazioni atomiche o lock**
3. **Far calcolare A := A[1] + A[2] da un terzo thread o nel main thread dopo il join dei due thread.**

Immagine che contiene testo, Carattere, schermata, ricevuta

Il contenuto generato dall'IA potrebbe non essere corretto.

Immagine che contiene testo, schermata, Carattere

Il contenuto generato dall'IA potrebbe non essere corretto.

🔐 Il costrutto atomic assicura che l'operazione di lettura, somma e scrittura su A sia **eseguita senza interferenze** da parte dell’altro thread.

📦 **Pro**:

* Semplice e sicura.
* Niente bisogno di lock complessi.

⚠️ **Contro**:

* Funziona bene solo per **operazioni semplici** (es. somme).
* Non tutte le architetture supportano **atomicità nativa** su operazioni complesse.

Immagine che contiene testo, Carattere, schermata

Il contenuto generato dall'IA potrebbe non essere corretto.

Immagine che contiene testo, schermata, Carattere, numero

Il contenuto generato dall'IA potrebbe non essere corretto.

* La sezione tra lock e unlock è **protetta**, quindi A non viene modificata da più thread contemporaneamente.

📦 **Pro**:

* Più generale: può proteggere anche **operazioni complesse** su strutture dati.
* Adatto anche in codice con più operazioni in sezione critica.

⚠️ **Contro**:

* Rischio di **deadlock** se i lock non sono usati correttamente.
* Potenziale **overhead** maggiore rispetto alle operazioni atomiche.

Immagine che contiene testo, schermata, Carattere, numero

Il contenuto generato dall'IA potrebbe non essere corretto.

Questo è un **esempio completo e corretto** di programmazione parallela con **memoria condivisa**, che mostra l'uso combinato di:

1. **Variabili private e condivise**
2. **Sezione critica con lock**
3. **Barriera di sincronizzazione**

**🧠 Perché funziona bene**

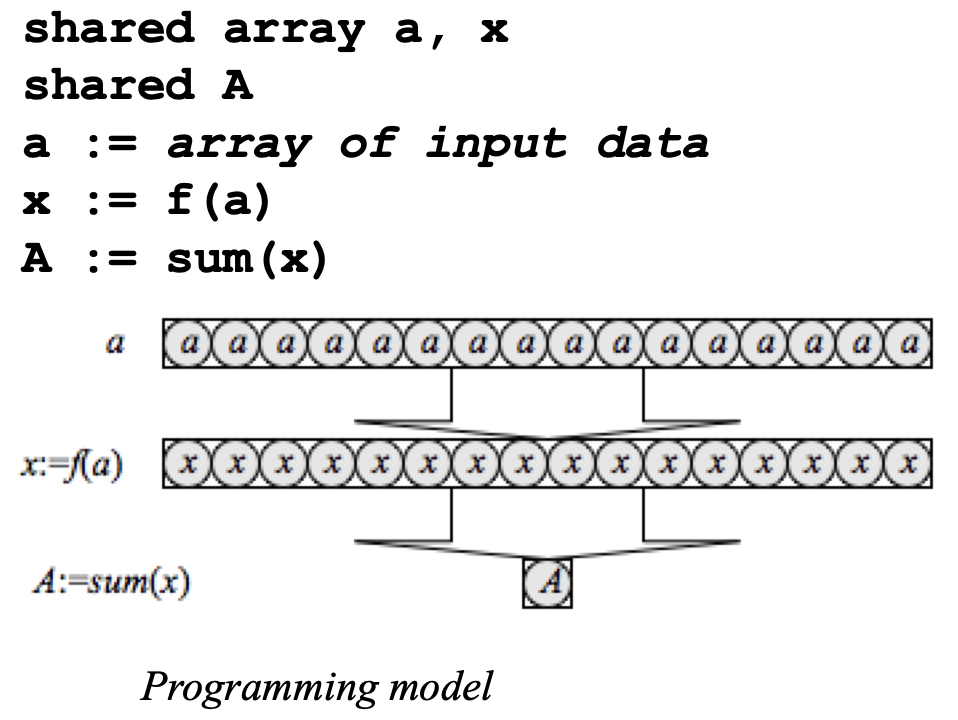
* ❌ **Nessuna race condition**: grazie al lock.
* ✅ **Corretto ordine di esecuzione**: garantito dalla barriera.
* 🚀 **Efficienza**: le somme locali vengono fatte in parallelo, e la somma finale è veloce (solo due accessi sincronizzati).

**🧩 Programming Model 2 – Data Parallel (SIMD) Programming**

**🔹 Idea centrale**

* Tutti i processori eseguono **la stessa istruzione**, ma **su dati diversi**.
* È ideale quando devi eseguire la stessa operazione su grandi quantità di dati (es. sommare tutti gli elementi di un array, moltiplicare matrici, ecc.).

**💻 Esempio tipico: Vector Machine**

****

* Un’unica istruzione viene applicata **a un intero vettore** di dati (es. array).
* Funziona in modo **pipelined**, cioè una nuova operazione parte prima che la precedente sia finita, come in una catena di montaggio.
* Ogni **elemento del vettore** è elaborato da un’unità distinta, ma con lo stesso codice.

**🖥️ SIMD Machine (moderna)**

* È formata da:
  + Un **processore di controllo centrale** (control processor).
  + Molti **processori semplici (elementary processors)** collegati in griglie scalabili.
* Esempi reali:
  + **MMX, SSE, AVX** su CPU moderne.
  + **GPU (Graphics Processing Units)** come CUDA e OpenCL.

**⚙️ Funzionamento**

* Il processore centrale **emette una singola istruzione**.
* Tutti i processori la eseguono **in parallelo su un pezzo di dati** (lock-step).
* Se c’è un’istruzione condizionale (if/else), **alcuni processori possono essere temporaneamente disattivati (masked)** se la condizione non vale → *predicated execution*

**🎯 Vantaggi**

* **Altissima efficienza** su dati uniformi.
* Perfetto per **elaborazione di segnali, immagini, video, AI**.
* Ottimo rapporto costo/prestazioni su carichi ben strutturati.

**⚠️ Limiti**

* Poco flessibile: difficile da usare quando il controllo del flusso diverge (es. if/else complicati).
* Meno adatto per algoritmi irregolari o con dipendenze dinamiche tra dati.

**✉️ Message Passing Programming**

Il **modello a passaggio di messaggi** è uno dei paradigmi fondamentali per la programmazione parallela, in particolare quando si lavora in **ambienti distribuiti** come **cluster HPC**, dove ogni nodo ha la propria memoria e il proprio spazio di esecuzione.

Immagine che contiene diagramma, schizzo, testo, Piano

Il contenuto generato dall'IA potrebbe non essere corretto.

**📌 Struttura del modello**

Nel message passing:

* Un **programma parallelo** è composto da **un insieme di processi**, ciascuno eseguito **su un nodo o un processore diverso** (tipicamente identificati come P0, P1, …).
* Ogni **processo** ha:
  + un **proprio thread di controllo** (quindi esegue in modo indipendente),
  + una **memoria locale privata** (non condivisa con altri),
  + un **indirizzamento locale** (non vede direttamente le variabili degli altri processi).

Poiché non esiste memoria condivisa, i processi **devono comunicare tra loro scambiandosi esplicitamente dei messaggi**.

**🔄 Comunicazione tra processi**

I processi si scambiano **messaggi punto-a-punto**, in cui:

* Uno è la **sorgente** (sender),
* L’altro è la **destinazione** (receiver).

Questi messaggi possono contenere:

* Dati da elaborare,
* Segnali di controllo o sincronizzazione.

Per farlo si usano tipicamente **librerie specializzate** come:

* **MPI** (Message Passing Interface) – lo standard più usato,
* **PVM** (Parallel Virtual Machine) – meno usato oggi ma storicamente importante.

**🌐 Architettura sottostante**

Ogni **nodo** o **processore** coinvolto ha una propria **interfaccia di rete**, che permette:

* Comunicazione con altri nodi,
* Sincronizzazione tra i processi (quando necessario).

Le **prestazioni della comunicazione** dipendono da:

* La **topologia di rete** (come sono connessi i nodi: a griglia, ad albero, torus, ecc.),
* Gli **algoritmi di routing** usati per indirizzare i messaggi tra sorgente e destinazione,
* La **latenza** (tempo per inviare un messaggio) e la **bandwidth** (quantità di dati trasmessi per secondo).

La **topologia di rete** può influenzare moltissimo la scalabilità ed efficienza del programma, come mostrato nella Figura 4 della presentazione.

**✅ Vantaggi del modello**

* Funziona bene in ambienti **distribuiti**, dove ogni nodo ha la sua memoria (es. supercomputer, cluster cloud).
* **Altamente scalabile**, adatto a sistemi con migliaia di processori.
* Consente una **chiara separazione dei dati**, evitando problemi come le **race condition**.

**⚠️ Svantaggi**

* È **più complesso da programmare** rispetto a un modello shared memory.
* Richiede una **gestione esplicita** della comunicazione e sincronizzazione.
* Le **prestazioni** dipendono fortemente dalla qualità della rete e dalla gestione dei messaggi.

Immagine che contiene testo, schermata, Carattere

Il contenuto generato dall'IA potrebbe non essere corretto.

Immagine che contiene testo, diagramma, Piano, schermata

Il contenuto generato dall'IA potrebbe non essere corretto.

Immagine che contiene testo, schermata, Carattere

Il contenuto generato dall'IA potrebbe non essere corretto.

Immagine che contiene testo, diagramma, Carattere, linea

Il contenuto generato dall'IA potrebbe non essere corretto.

Immagine che contiene testo, schermata, Carattere

Il contenuto generato dall'IA potrebbe non essere corretto.

Immagine che contiene diagramma, linea, Piano, Disegno tecnico

Il contenuto generato dall'IA potrebbe non essere corretto.

Immagine che contiene testo, schermata, Carattere, numero

Il contenuto generato dall'IA potrebbe non essere corretto.

**🔗 Hybrid Systems: la combinazione di più modelli di parallelismo**

Nei moderni **sistemi di calcolo ad alte prestazioni (HPC)**, è sempre più comune utilizzare **architetture ibride**, ovvero **cluster di nodi SMP (Symmetric MultiProcessing)**. In pratica, ogni nodo è composto da **più processori** che **condividono la stessa memoria**, ma i nodi sono tra loro **separati** e comunicano tramite rete.

Questa complessità architetturale impone di adottare **modelli di programmazione ibridi**, che combinano tecniche diverse per ottenere **prestazioni ottimali** e **scalabilità**. Nella pratica, ci sono tre principali approcci.

**🟦 1. Flat model – Solo *message passing***

Nel modello “piatto”, si **ignora la gerarchia interna dei nodi SMP** e si usa solo il **message passing (es. MPI)** per tutta la comunicazione, **anche tra processi sullo stesso nodo**.

**✅ Vantaggi:**

* **Semplice da programmare**: un solo modello (MPI) per tutti i livelli.
* **Alta portabilità**: il codice può essere eseguito su cluster SMP, distribuiti, cloud, ecc.

**❌ Svantaggi:**

* Si **perdono i benefici** della memoria condivisa interna ai nodi (UMA).
* Maggiore **overhead comunicativo**, anche dove sarebbe evitabile.

**🟨 2. Two-layer model – Shared memory + message passing**

Qui si usa un approccio **ibrido**:

* **All’interno di ogni nodo SMP**: si sfrutta la **shared memory** (tipicamente con OpenMP).
* **Tra nodi diversi**: si utilizza il **message passing** (tipicamente con MPI).

**✅ Vantaggi:**

* Ottimo compromesso tra **prestazioni** e **flessibilità**.
* Si sfrutta la **shared memory locale** per ridurre il traffico di rete.
* È il modello **più usato oggi nei cluster HPC**.

**❌ Svantaggi:**

* **Più complesso da programmare**: bisogna gestire due paradigmi distinti.
* La suddivisione del lavoro deve essere progettata **in modo consapevole** per non creare colli di bottiglia.

**🟥 3. Three-layer model – SIMD + shared memory + message passing**

Questo è il modello **più sofisticato**, che combina:

* **SIMD** a livello di **singolo core** (es. istruzioni AVX, SSE o GPU).
* **Shared memory programming** tra core sullo **stesso nodo SMP**.
* **Message passing** tra nodi diversi nel cluster.

**✅ Vantaggi:**

* Massimizza l’efficienza a **tutti i livelli di parallelismo**.
* Ottimale per architetture con **GPU**, **coprocessori**, o sistemi con molti core.

**❌ Svantaggi:**

* **Molto difficile da implementare**: richiede conoscenze approfondite di tutti i modelli.
* Bisogna scrivere codice ad hoc per ogni livello (SIMD, OpenMP, MPI).

📌 **Esempio reale**: Un'applicazione scientifica può usare:

* **CUDA/OpenCL** per il calcolo su GPU (SIMD),
* **OpenMP** per parallelo tra core CPU,
* **MPI** per distribuire i dati tra i nodi.

**⏱️ Bulk Synchronous Processing (BSP)**

Il **modello BSP** è un approccio alla programmazione parallela che cerca di offrire un **buon equilibrio tra semplicità, efficienza e prevedibilità**. È stato introdotto da **Leslie Valiant** per semplificare la scrittura e la comprensione di programmi paralleli, specialmente in ambito **data-parallel**.

A differenza di modelli più liberi (come MIMD puro), BSP **impone una struttura ben definita** all'esecuzione parallela, dividendola in unità chiamate **superstep**.

**🔁 Struttura del BSP: i *superstep***

Ogni **programma parallelo** in BSP è organizzato come una **sequenza di superstep**, e ogni superstep ha sempre tre fasi:

1. **Compute phase**  
   Ogni processo o nodo **lavora sui dati locali**. In questa fase avviene la maggior parte del calcolo computazionale. Se si è su un’architettura **shared memory (SMP)**, è possibile anche **leggere dati** dalla memoria condivisa, ma ogni processo opera indipendentemente.
2. **Communication phase**  
   Una volta finito il calcolo, tutti i processi **scambiano dati tra di loro**, oppure eseguono una **riduzione di dati globali** (come sommare tutti i risultati parziali). La comunicazione è **sincronizzata**, non avviene in tempo reale durante il calcolo ma in una fase apposita.
3. **Barrier synchronization**  
   Alla fine del superstep c’è una **barriera di sincronizzazione**: tutti i processi **devono attendere** che gli altri abbiano terminato prima di passare al superstep successivo. Questo meccanismo **garantisce coerenza e ordine** nell’esecuzione.

**✅ Vantaggi**

* **Semplicità concettuale**: la suddivisione rigida in fasi rende più facile prevedere cosa accade in ogni momento.
* **Prevedibilità delle performance**: si può stimare il tempo di esecuzione in modo più controllato.
* **Ideale per data-parallel computing**, dove ogni processo lavora su una porzione dei dati.

**❗ Svantaggi**

* Le **barriere obbligatorie** tra superstep possono introdurre **latenza**: se alcuni processi finiscono prima, devono comunque aspettare gli altri.
* Meno flessibile rispetto a modelli asincroni come MIMD puro.
* La comunicazione tra superstep può essere **costosa**, specialmente in ambienti distribuiti.

Immagine che contiene testo, schermata, Carattere

Il contenuto generato dall'IA potrebbe non essere corretto.

Immagine che contiene testo, schermata, Carattere, numero

Il contenuto generato dall'IA potrebbe non essere corretto.

**02-FCAPP-Code\_Offloading**

**🚀 Cos’è il Code Offloading?**

**Code offloading** è la tecnica di **spostare parte del carico computazionale** da un **processore principale (CPU)** a un **acceleratore esterno**, come:

* una **GPU** (Graphics Processing Unit),
* un **coprocessore many-core** (es. Intel MIC/Xeon Phi),
* oppure un altro **nodo remoto** (nel cloud o in un cluster).

**🧠 Perché si fa?**

* Per **velocizzare l'esecuzione** di sezioni di codice molto pesanti (es. moltiplicazione di matrici, algoritmi scientifici, reti neurali).
* Per **risparmiare energia** (gli acceleratori spesso hanno un miglior rapporto FLOP/watt).
* Per **sfruttare l’hardware disponibile** (molti cluster e cloud hanno GPU installate).

**🚀 Compute Performance (Prestazioni computazionali)**

**✅ Promessa**

* Le **GPU** e i **coprocessori many-core** (tipo Intel Xeon Phi) offrono **altissime prestazioni teoriche**.
* Hanno centinaia o migliaia di core → adatti per **operazioni parallele massicce**.

**⚠️ Caveat (attenzione!)**

* Nei casi reali (app scientifiche complesse), il **guadagno effettivo è spesso più basso**:
  + Invece di un 10×, ci si accontenta di **2x o 3x rispetto a una CPU multicore**.
  + Questo dipende da: **trasferimento dati, struttura degli algoritmi, accessi in memoria**.

**📌 Osservazioni**

* Esistono casi spettacolari di successo (es. **performance da PetaFlops** → Gordon-Bell Prize).
* Tuttavia, il marketing (Intel/NVIDIA) è stato **molto aggressivo**, e oggi si ha un approccio **più realistico e critico**.

**🔋 Energy Efficiency (Efficienza energetica)**

**✅ Promessa**

* Le GPU consumano **meno energia per FLOP** (operazione in virgola mobile).
* Questo è **fondamentale per i supercomputer exascale**, dove il consumo è un limite critico.

**⚠️ Caveat**

* Anche qui, le **applicazioni reali** in ambienti ibridi CPU-GPU spesso **non mantengono l’efficienza ideale**:
  + Perché i dati si devono spostare (PCIe), o
  + perché ci sono fasi seriali o mal distribuite.

**🌍 Existing Resources (Risorse disponibili)**

**✅ Situazione attuale**

* Oggi ci sono **tantissimi sistemi già dotati di GPU e many-core**.
* In molti ambiti (HPC, AI, Big Data), la tecnologia è vista come **inevitabile**.

**⚠️ Caveat**

* Il vero ostacolo è: **quanto costa scrivere, portare e mantenere applicazioni efficienti per queste architetture?**
  + **Code offloading** richiede competenze, test, ottimizzazioni.
  + Il **tempo e l’effort umano** sono significativi.

**🧠 2.1 NVIDIA GPU Technology – Programmare su CPU e GPU**

Quando si lavora con le **GPU NVIDIA**, il modello di programmazione è **eterogeneo**, cioè prevede la **suddivisione del codice** in due parti:

* La **host code**, che viene eseguita sulla **CPU**;
* La **device code**, che viene eseguita sulla **GPU**.

**Come funziona?**

* La CPU ha il **controllo principale** del programma: gestisce il flusso, le decisioni, la logica.
* La GPU si occupa delle **parti più pesanti dal punto di vista computazionale**, quelle adatte a essere eseguite in parallelo, come calcoli su grandi insiemi di dati.

Immagine che contiene testo, schermata, Carattere, ricevuta

Il contenuto generato dall'IA potrebbe non essere corretto.

Immagine che contiene schermata, testo, diagramma, Rettangolo

Il contenuto generato dall'IA potrebbe non essere corretto.

**🧩 Struttura interna**

**✅ CPU: ottimizzata per la flessibilità e la risposta rapida**

* Pochi **core molto complessi**, con:
  + **Cache di livello 2 (e 3)** molto grande
  + Unità di controllo potente
  + ALU (unità logiche-aritmetiche; è il circuito che **esegue le operazioni matematiche e logiche** fondamentali.)
* Accede alla memoria secondo il **principio di località**:
  + Accesso **rapido** ai dati più usati di recente → **bassa latenza**
* Ideale per:
  + **Task seriali o ramificati**
  + Logica, flusso condizionale, I/O, sistemi operativi

**✅ GPU: ottimizzata per il calcolo massivo parallelo**

* **Tantissimi core semplici** (centinaia/migliaia), con:
  + Piccola cache L2
  + Nessuna o limitata unità di controllo
* Pensata per eseguire **la stessa operazione su molti dati** → **altissimo throughput**
* Ideale per:
  + **Simulazioni numeriche**
  + Elaborazione immagini, reti neurali, video encoding

N.B: Latenza: È il **tempo di attesa** tra il momento in cui chiedi qualcosa e il momento in cui ottieni una risposta.

Throughput: È la **quantità di lavoro completato in un intervallo di tempo**. In altre parole: **quante operazioni riesci a fare ogni secondo**.

**Caratteristica fondamentale delle GPU:**

Il **flusso di esecuzione deve essere omogeneo**, cioè tutti i thread devono eseguire **le stesse istruzioni**. Questo perché l’architettura delle GPU segue il paradigma **SIMT** (Single Instruction, Multiple Threads), simile a SIMD: se c’è un “if” che divide i thread in strade diverse, la GPU rallenta.

**Limite del sistema: il PCIe**

I dati tra CPU e GPU passano attraverso il **bus PCIe**, che è relativamente **lento** rispetto alla velocità con cui la GPU può lavorare sui dati già presenti nella sua memoria. Questo rappresenta un **collo di bottiglia**: è importante sovrapporre la comunicazione e la computazione per minimizzare l’impatto.

**Linguaggi e strumenti**

Per programmare le GPU NVIDIA si usano principalmente:

* **CUDA**, linguaggio proprietario NVIDIA;
* **OpenCL**, linguaggio aperto e multipiattaforma.

I kernel CUDA (o OpenCL) sono scritti in **C o C++** e compilati per essere eseguiti sulla GPU. La parte host è anch’essa in C/C++, e controlla l’avvio dei kernel, la copia dei dati tra CPU e GPU, e la gestione della memoria.

A supporto, NVIDIA mette a disposizione **strumenti di sviluppo** per:

* Debugging,
* Profiling,
* System monitoring.

Il tutto è pensato per essere **ad alto livello**, simile a quello che si fa con **OpenMP**, ma in ambiente eterogeneo.

**⚙️ 2.2 CPU vs GPU – Differenze architetturali**

**CPU: generalista e a bassa latenza**

La **CPU (Central Processing Unit)** è il cervello generale del computer. È progettata per eseguire una vasta gamma di istruzioni in modo **veloce e flessibile**, ma **non massivamente parallelo**.

Caratteristiche principali:

* **Cache L2 molto grande**: immagazzina dati frequentemente usati, sfruttando la **località spaziale e temporale** (cioè tende a usare di nuovo dati vicini a quelli appena usati).
* **ALU (unità di elaborazione aritmetica)** e **unità di controllo** che permettono alta efficienza nel gestire task sequenziali e logica complessa.
* Ottimizzata per **bassa latenza**: i dati vengono recuperati molto rapidamente.

Risultato: la CPU è **veloce su pochi task**, ma non riesce a gestire bene operazioni ripetitive e molto parallele su grandi moli di dati.

**GPU: specialista ad alto throughput**

La **GPU (Graphics Processing Unit)** è nata per la grafica, ma oggi è usata per moltissimi calcoli (da cui il termine **GPGPU** – General-Purpose GPU).

Caratteristiche:

* Contiene **molti più core** della CPU, anche centinaia o migliaia.
* **Poca cache**: è meno efficiente per operazioni che richiedono accessi a dati complessi o non locali.
* Ottimizzata per **throughput alto**, cioè per eseguire **tanti calcoli in parallelo**, anche se singolarmente più lenti.

Questo la rende perfetta per compiti **data-parallel**, come:

* Operazioni su matrici e vettori,
* Rendering grafico,
* Calcolo scientifico e simulazioni.

**🚀 CUDA Programming Model: Come lavora la GPU**

Nel modello CUDA, la **GPU viene vista come un coprocessore**, cioè una componente aggiuntiva che affianca la CPU nell’elaborazione. Questo coprocessore è **dotato di una propria memoria** e può essere utilizzato per eseguire parti del programma che richiedono **molta potenza di calcolo**, in particolare operazioni **data-parallel**, cioè su grandi insiemi di dati elaborati in parallelo.

Le porzioni di codice destinate alla GPU vengono scritte in funzioni speciali chiamate **kernel CUDA**. Questi kernel sono progettati per essere eseguiti **contemporaneamente da molti thread**, ognuno dei quali lavora su un pezzetto diverso dei dati, in modo **indipendente**. Questo approccio segue il paradigma **SPMD (Single Program, Multiple Data)** e in particolare il modello **SIMT (Single Instruction, Multiple Threads)** definito da NVIDIA: tutti i thread eseguono lo stesso programma, ma su dati diversi.

**⚙️ CUDA Execution Model: Come funziona in pratica**

Il **flusso di lavoro** tra CPU e GPU si svolge in **quattro fasi principali**:

1. **Copia dei dati** dalla memoria della **CPU** a quella della **GPU** tramite il bus PCIe.
2. **Caricamento ed esecuzione del kernel** CUDA sulla GPU in modo massivamente parallelo.
3. **Scrittura dei risultati** nella memoria della GPU.
4. **Copia dei risultati** dalla GPU alla memoria della CPU.

Questa separazione è necessaria perché CPU e GPU hanno **spazi di memoria separati**. Tuttavia, le versioni più recenti delle GPU permettono di **automatizzare** o semplificare parte di questo processo, rendendolo più **trasparente per il programmatore**.

**🔄 Come viene eseguito un kernel CUDA**

Quando viene lanciato un kernel CUDA, i **thread** sono raggruppati in **blocchi** (thread blocks). Ogni blocco viene assegnato a uno degli **Streaming Multiprocessor (SM)** della GPU in modo **round-robin**. A seconda delle **risorse disponibili** (registri, memoria condivisa, ecc.), un SM può gestire un certo numero di blocchi.

I blocchi vengono gestiti da una **coda di esecuzione**:

* Quando un SM termina l'esecuzione di un blocco, ne riceve un altro dalla coda.
* I blocchi sono **indipendenti tra loro**: non possono comunicare né sincronizzarsi direttamente.

All’interno di ogni blocco:

* I thread sono organizzati in **warp** da 32 thread.
* Ogni warp viene eseguito in modo **sincrono**, cioè i 32 thread eseguono la **stessa istruzione alla volta**.
* Se i thread prendono **strade diverse** (es. un if con condizioni diverse), l’esecuzione diventa **meno efficiente**, perché lo scheduler CUDA deve eseguire le diverse istruzioni separatamente (branch divergence).

**🔁 Scalabilità trasparente**

Uno dei vantaggi principali di CUDA è la sua **scalabilità trasparente**:

Immagine che contiene testo, schermata, linea, Rettangolo

Il contenuto generato dall'IA potrebbe non essere corretto.

* Lo **stesso codice** può girare su GPU diverse con **numeri differenti di SM**.
* Il sistema CUDA si occupa **automaticamente** di distribuire i blocchi sui vari SM, senza che il programmatore debba modificare il codice.
* Questo rende il modello **estremamente portabile e adattabile**, mantenendo buone prestazioni.

**🌐 Heterogeneous High Performance Programming Framework**

In ambito HPC (High Performance Computing), **CUDA** e **OpenCL** sono i due strumenti principali per programmare in modo parallelo su GPU. **CUDA**, sviluppato da NVIDIA, è stato a lungo il framework dominante, soprattutto grazie alle sue performance e all'integrazione con le GPU NVIDIA. È molto diffuso, ben supportato, e offre strumenti avanzati per debugging, profiling, e gestione delle risorse.

Tuttavia, **OpenCL (Open Computing Language)** si è ormai evoluto e maturato fino al punto da essere considerato una valida alternativa, soprattutto per chi cerca **portabilità**. A differenza di CUDA, OpenCL è uno **standard aperto**, mantenuto da **Khronos Group**, ed è **compatibile con molte architetture hardware**, comprese GPU di AMD, Intel e NVIDIA, ma anche CPU, DSP (Digital Signal Processor) e altri acceleratori, inclusi dispositivi mobili o embedded.

Immagine che contiene testo, schermata, Carattere, numero

Il contenuto generato dall'IA potrebbe non essere corretto.

**🧠 Perché OpenCL è importante nell’HPC moderno?**

Oggi i sistemi di calcolo non sono più composti solo da CPU, ma da un insieme **eterogeneo di risorse**: CPU, GPU, acceleratori specializzati (come TPU o FPGA), DSP e altro. OpenCL nasce proprio per gestire **queste architetture eterogenee** con **un singolo programma portabile**, scrivibile una volta sola ma eseguibile ovunque.

Con OpenCL, è possibile **sfruttare tutte le risorse disponibili in una piattaforma**: ad esempio, una simulazione può assegnare alcuni calcoli alla CPU, altri alla GPU, e altri ancora a un DSP, in base alle caratteristiche e alle necessità. Questo lo rende molto potente per applicazioni scientifiche, ingegneristiche, simulazioni fisiche e analisi dati.

**🧩 In sintesi**

Il **futuro del calcolo ad alte prestazioni** richiede strumenti flessibili, scalabili e portabili. **CUDA** resta uno standard industriale fortissimo per chi lavora esclusivamente con hardware NVIDIA e cerca la massima ottimizzazione. **OpenCL**, invece, si propone come **scelta strategica per ambienti eterogenei**, dove è importante poter eseguire il codice su qualunque tipo di dispositivo.

Questa **apertura e versatilità** fanno di OpenCL un pilastro emergente del calcolo parallelo, sempre più considerato anche da chi lavora in ambito scientifico, HPC e AI.

**🧩 OpenCL: La Controparte Aperta di CUDA**

**OpenCL** (Open Computing Language) è l’alternativa **non proprietaria** e **open standard** a CUDA, progettata per offrire un'interfaccia di programmazione comune per una vasta gamma di dispositivi eterogenei: **GPU AMD**, **CPU** di diversi produttori, **FPGA**, **MIC**, e molto altro. Questo lo rende uno strumento estremamente **portabile**, ideale in contesti in cui si vuole evitare il legame con un singolo vendor, come NVIDIA.

Tuttavia, come CUDA, **OpenCL è un framework di basso livello**, e per questo motivo richiede **competenze avanzate di programmazione** per essere utilizzato in modo efficace. Non è pensato per sviluppatori alle prime armi, ma per chi ha bisogno di **massimo controllo** e **flessibilità** nelle architetture eterogenee.

**🧱 OpenCL Platform Model**

L’architettura di OpenCL è composta da due entità principali:

* **Host**: tipicamente la CPU che gestisce il programma.
* **OpenCL Device**: uno o più dispositivi (GPU, CPU, FPGA, ecc.) su cui viene eseguito il calcolo parallelo.

Ogni OpenCL device è ulteriormente suddiviso in:

* **Compute Units (CU)**: unità di calcolo. Nelle GPU, corrispondono agli Streaming Multiprocessor (SM).
* **Processing Elements (PE)**: elementi di elaborazione all’interno delle CU. Ciascuno esegue effettivamente le istruzioni sui dati.

Immagine che contiene diagramma, testo, Piano, linea

Il contenuto generato dall'IA potrebbe non essere corretto.

🧠 **Per le CPU**, si ha:

* Una CU per core.
* Una PE per CU, oppure più PE se i core usano SIMD.

🎮 **Per le GPU**, ciascun dispositivo è un device OpenCL distinto:

* Una CU per ogni SM.
* Diversi PE per ogni CU, per supportare l'elevato parallelismo.

📦 Inoltre, la memoria è organizzata in:

* **Host Memory** (gestita dalla CPU).
* **Device Memory** (associata ai dispositivi acceleratori).

**⚙️ Inizializzazione e Uso**

A differenza di **CUDA**, che **inizializza automaticamente** la GPU e offre anche funzionalità avanzate gestibili manualmente, **OpenCL richiede sempre un’inizializzazione esplicita**. Questo perché non si lavora solo con GPU NVIDIA, ma si può utilizzare **qualsiasi tipo di dispositivo supportato**, per cui è necessario specificare in maniera dettagliata **quale device usare**.

In OpenCL, la gestione è molto dettagliata (e quindi più **prolissa**): tutto deve essere specificato e gestito manualmente. Questo può sembrare uno svantaggio, ma in realtà permette una maggiore **flessibilità** e **controllo**, e soprattutto garantisce **portabilità** del codice.

💡 Tuttavia, grazie alla struttura modulare di OpenCL, è possibile scrivere **funzioni riutilizzabili** per incapsulare le operazioni comuni (come la gestione dei buffer, il lancio dei kernel, ecc.), rendendo il codice più manutenibile e leggibile.

**🧠 Conclusione**

OpenCL è uno strumento **potente, portabile e flessibile**, pensato per chi lavora su architetture **eterogenee**. Sebbene richieda una curva di apprendimento più ripida rispetto a CUDA, rappresenta una soluzione strategica per chi ha bisogno di **massima compatibilità e astrazione hardware**, soprattutto in contesti HPC e embedded.

**Shared Memory su GPU**

All’interno di una GPU, è possibile utilizzare una memoria condivisa tra i **thread di uno stesso blocco** per **ottimizzare le prestazioni** e **ridurre l'accesso alla global memory**, che è molto più lenta.

Questa **shared memory** è una porzione di memoria **veloce** che consente ai thread dello stesso blocco di **cooperare** e **scambiarsi dati** in modo efficiente. Il vantaggio principale è che un thread può **riutilizzare dati** che un altro thread ha già caricato dalla **global memory** nella **shared memory**, **evitando duplicazioni** di accessi lenti.

**🧩 Workflow tipico con la shared memory:**

1. 🔧 **Dichiarazione di un buffer** nella shared memory (ad esempio un array condiviso).
2. 📥 **Caricamento dei dati** dalla global memory nel buffer da parte di uno o più thread.
3. ⏱️ **Sincronizzazione dei thread del blocco**: prima di procedere oltre, è necessario essere certi che tutti i dati siano stati caricati correttamente nel buffer.
4. 🧮 **Elaborazione dei dati**: ogni thread lavora sui dati condivisi, magari con operazioni parallele e indipendenti.
5. ✅ **Nuova sincronizzazione**: per garantire che tutti i thread abbiano terminato le operazioni prima di andare avanti.
6. 💾 **Scrittura dei risultati finali** nella global memory, che poi sarà accessibile anche ad altri blocchi o all’host.

**🔍 Esempio pratico:**

Supponiamo di voler eseguire un’operazione di somma tra elementi di un array. Invece di far sì che ogni thread acceda direttamente alla **global memory** ogni volta (operazione lenta), possiamo:

* caricare l’array in **shared memory** una sola volta,
* far lavorare i thread localmente,
* e infine scrivere il risultato nella global memory.

**🎯 Vantaggi chiave:**

* **Riduzione dei tempi di accesso alla memoria**
* **Maggiore efficienza computazionale**
* **Perfetto per operazioni ripetitive su dati locali**

La shared memory è quindi **un’arma fondamentale per migliorare le performance** su architetture CUDA o OpenCL, e sfruttarla correttamente può fare la differenza nei contesti ad alta intensità di calcolo come quelli dell’HPC.

Immagine che contiene testo, schermata, diagramma, Rettangolo

Il contenuto generato dall'IA potrebbe non essere corretto.

**🎯 Perché è importante usare la local memory?**

* Perché la **global memory è lenta** (accessi in centinaia di cicli).
* La **shared memory è veloce** (accessi in pochi cicli).
* Se più thread hanno bisogno **degli stessi dati**, è **molto più efficiente** caricarli una sola volta in shared memory.

**🔧 Quando e perché usarla**

✅ **Usa Local Memory se**:

* Più thread **devono riutilizzare gli stessi dati** → eviti di leggerli più volte dalla global memory.
* Hai bisogno di **alta velocità** e puoi **controllare il flusso di accesso**.

🚫 **Non usarla se**:

* Sei su CPU o Intel MIC: **non c'è hardware dedicato**, quindi può **rallentare**.
* La GPU ha **cache automatica efficace** (in GPU moderne) → a volte **non serve più** ottimizzare a mano.

**03-FCAPP-Computational\_Malleability\_meet\_FlexMPI**

**🔹 Rigid Job**

Il **rigid job** è il tipo più semplice e comune nei sistemi di calcolo parallelo. Viene lanciato con un numero fisso di risorse (ad esempio un certo numero di nodi o processi MPI), specificato al momento della sottomissione. Questo numero **non può essere modificato** né prima dell’esecuzione né durante il runtime. È una scelta molto diffusa perché è facile da gestire e compatibile con quasi tutti i sistemi di scheduling, ma ha lo svantaggio di essere completamente **invariabile**: se le risorse richieste non sono disponibili, il job resta in coda in attesa, senza possibilità di adattarsi.

Immagine che contiene testo, schermata, Carattere, Blu elettrico

Il contenuto generato dall'IA potrebbe non essere corretto.

**🔹 Moldable Job**

Immagine che contiene testo, schermata, diagramma, Carattere

Il contenuto generato dall'IA potrebbe non essere corretto.Il **moldable job** introduce un primo livello di flessibilità. In questo caso, il job viene sottomesso specificando **un intervallo di risorse accettabili** (ad esempio: minimo 8, massimo 32 nodi). Sarà il sistema batch a decidere, **prima dell’esecuzione**, quante risorse assegnare, in base a quelle disponibili. Una volta avviato, però, il job si comporta come un rigid job: **non può più essere riconfigurato**. Questo approccio è utile per sfruttare al meglio le risorse libere nei momenti di minor carico, ma non permette un vero adattamento dinamico.

**Resource Management System (RMS)**

In ambito HPC e sistemi operativi, l’**RMS** è: **Sistema di Gestione delle Risorse**.

È un componente software che si occupa di:

* Allocare CPU, memoria, dischi o GPU tra i vari job o utenti.
* Gestire la **priorità**, la **distribuzione** e l’**esecuzione dei processi**.
* Interfacciarsi con lo scheduler (es. **SLURM**, PBS, ecc.).

**🔹Evolving Job**

L’**evolving job** è capace di adattarsi **durante l’esecuzione**, ma solo **aumentando** il numero di risorse utilizzate. Questo significa che il job può chiedere, in fase di runtime, nodi o processi aggiuntivi al sistema batch. Questo può accadere, ad esempio, se emergono nuove necessità computazionali, o se aumentano i dati da elaborare. Il job può quindi “evolvere” in risposta a ciò che scopre mentre lavora. Tuttavia, **non è in grado di ridurre** il numero di risorse, né di riallocarsi in caso di carenza: la sua adattività è **solo verso l’alto**.

Immagine che contiene testo, schermata, diagramma, Carattere

Il contenuto generato dall'IA potrebbe non essere corretto.

**🔹 Malleable Job**

Il **malleable job** rappresenta una forma ancora più avanzata di flessibilità. Può **espandersi o ridursi dinamicamente** durante l’esecuzione, in base alla disponibilità delle risorse o alle decisioni del sistema. Se, ad esempio, diventano disponibili più core o nodi, il sistema può assegnarli al job, che li userà immediatamente; viceversa, se serve liberare risorse per altri job più urgenti, il sistema può sottrarle, e il job si adatta riducendo il parallelismo. Questo richiede un’infrastruttura avanzata, come **FlexMPI** o **OpenMP**, in grado di gestire questi cambiamenti **in modo trasparente**. È molto utile per migliorare l’uso delle risorse e ridurre i tempi di attesa in ambienti condivisi.

Immagine che contiene testo, schermata, diagramma, Carattere

Il contenuto generato dall'IA potrebbe non essere corretto.

**🔹 Adaptive Job**

Il **adaptive job** è il più intelligente e autonomo. Non solo può modificare dinamicamente il numero di risorse come un malleable, ma **decide attivamente quando farlo**, reagendo a eventi esterni o condizioni del sistema. Ad esempio, se il job viene sospeso per motivi di scheduling, può ricordare quali risorse stava usando al momento dell’interruzione e richiederle di nuovo al restart. Questo tipo di job **collabora attivamente con il sistema batch**, adattandosi a carichi variabili e migliorando sia l’efficienza dell’applicazione che l’utilizzo complessivo delle risorse.

Immagine che contiene testo, schermata, diagramma, Carattere

Il contenuto generato dall'IA potrebbe non essere corretto.

Immagine che contiene testo, schermata, Carattere, numero

Il contenuto generato dall'IA potrebbe non essere corretto.

**🔍 Cos’è FlexMPI – Panoramica**

**FlexMPI** è un’estensione della libreria **MPI (Message Passing Interface)** pensata per portare funzionalità di **malleabilità** e **adattività dinamica** alle applicazioni parallele.  
È progettata per ambienti **non dedicati** o **eterogenei**, dove le risorse cambiano nel tempo (es. cloud, cluster condivisi).

A differenza di MPI “standard”, che è rigido, FlexMPI **monitora continuamente le prestazioni dell’applicazione** e usa queste informazioni per prendere decisioni sul **carico di lavoro**, sulla **distribuzione dei dati** e sull'eventuale **adattamento del numero di processi**.

L’obiettivo principale è **adattare in tempo reale** il numero di processi MPI in base a:

* disponibilità delle risorse,
* prestazioni misurate a runtime,
* politiche definite dall’utente o dal sistema.

**🧩 Caratteristiche principali di FlexMPI**

**🟩 1. Malleabilità e Adattività**

* Supporta job **malleabili**: il numero di processi può **cambiare durante l’esecuzione**.
* Supporta job **adattivi**: il comportamento si adatta in base a **politiche predefinite**:
  + Massime prestazioni
  + Efficienza energetica
  + Costo d'esecuzione
  + Politiche personalizzate (user-defined)

**🟩 2. Monitoring e Load Balancing**

* Integra **monitoraggio delle performance** con **hardware counters** (prestazioni misurate direttamente dall’hardware).
* Supporta **bilanciamento del carico** su:
  + Sistemi dedicati
  + Sistemi condivisi (non dedicati, ad es. cluster condivisi o cloud)

**🟩 3. Controllo centralizzato tramite Controller**

* Il **controller esterno** comunica con l'applicazione MPI per:
  + Gestire **la schedulazione dei job**
  + Gestire **le operazioni I/O** (grazie a CLARISSE)
  + Applicare decisioni dinamiche (es. ridurre processi se sistema è sovraccarico)

**🧩 Struttura di FlexMPI**

FlexMPI è composto da tre elementi principali:

1. **Una libreria multithread** inserita all’interno di ogni applicazione MPI, che intercetta le chiamate di comunicazione (MPI\_Send, MPI\_Recv, ecc.) e monitora i dati (tramite *wrapper*).
2. **Un controller esterno**, che coordina l’esecuzione e può anche gestire più applicazioni simultaneamente.
3. **Un’interfaccia di integrazione** con sistemi esterni come **scheduler**, **controller intelligenti** (es. IC in ADMIRE) e **sistemi di I/O** (es. Hercules, CLARISSE).

**🧠 Requisiti delle applicazioni**

FlexMPI è pensato per applicazioni:

* basate su **MPI**,
* scritte secondo il modello **SPMD** (Single Program, Multiple Data),
* **iterative**, cioè che alternano **fasi di calcolo e comunicazione**.

Esempi: **Jacobi**, **Gradiente Coniugato**, **motori Lagrangiani**.

**📈 Schedulazione e I/O**

Il **controller** di FlexMPI non si occupa solo della malleabilità “classica” (processi MPI), ma anche di:

* **Application scheduling**: decide come e quando eseguire o riconfigurare un'applicazione.
* **I/O scheduling**: può collegarsi a sistemi come **Hercules** o **CLARISSE** per decidere dinamicamente quanti nodi dedicare al **file system**, in base alle necessità di I/O (input/output).

**🔄 Come funziona la malleabilità**

Durante l’esecuzione:

* Il controller può **attivare politiche** basate su metriche (prestazioni, tempo, carico).
* L'app può **aumentare o diminuire i processi MPI**.
* FlexMPI **ridistribuisce automaticamente i dati** tra i nuovi processi, mantenendo la coerenza.
* La ridistribuzione avviene su **strutture dati registrate** in FlexMPI (matrici, vettori densi o sparsi).

**⚖️ Load Balancing in azione**

Il **load balancing** viene attivato:

* Quando vengono **creati o terminati** processi.
* Quando i **nodi hanno capacità computazionali diverse** (es. un cluster eterogeneo).
* Quando si lavora su **infrastrutture condivise**, dove le prestazioni possono variare nel tempo.

**🔁 Comunicazioni MPI “wrappate”**

FlexMPI **non sostituisce MPI**, ma **intercetta (wrap)** le chiamate standard come MPI\_Bcast, MPI\_Scatter, MPI\_Gather, ecc. Questo permette di aggiungere funzionalità avanzate **senza modificare il codice legacy**: l’app può usare MPI normalmente, ma dietro le quinte FlexMPI raccoglie dati e gestisce il runtime.

**🧠 Quando è utile FlexMPI?**

Ci sono due contesti principali dove la malleabilità di FlexMPI è particolarmente vantaggiosa:

1. **Cluster dedicati esclusivamente a job malleabili**: evita sprechi di risorse, ma bisogna impedire che job non malleabili occupino risorse staticamente.
2. **Cluster (anche in cloud) con risorse limitate o on-demand**: la malleabilità permette di ottimizzare i costi e adattarsi alla disponibilità.

**🧠 FlexMPI Controller – Cosa fa**

Il **Controller** è il componente **esterno e centrale** che coordina l'esecuzione di più applicazioni MPI malleabili in parallelo. È responsabile della **gestione dinamica** delle risorse durante il runtime.

Ecco le sue funzioni principali:

* ✅ **Avvia e controlla più applicazioni MPI** contemporaneamente.
* 📤 **Invia comandi di riconfigurazione** (es. aggiungi/rimuovi processi).
* 📥 **Riceve i dati di monitoraggio** da ogni applicazione in esecuzione.
* 🛑 **Termina le applicazioni** quando richiesto.

Il Controller comunica con le app via un’**API interna FlexMPI**, usando un sistema di comandi numerici.

Immagine che contiene testo, schermata, Carattere, numero

Il contenuto generato dall'IA potrebbe non essere corretto.

**🧩 Come si integra nel sistema**

Il **Controller** lavora insieme a:

* La **libreria FlexMPI**, presente in ogni processo MPI.
* Il sistema di schedulazione (es. Slurm).
* Altri tool come **Hercules** o **CLARISSE**, se si vuole controllare anche l’I/O.

È lo strumento che **rende possibile l’adattività**: senza Controller, FlexMPI non saprebbe **quando e come riconfigurare l’applicazione**.

**🔁 FlexMPI – Control Flow**

**Immagine che contiene testo, schermata, Carattere, diagramma

Il contenuto generato dall'IA potrebbe non essere corretto.**

**🧩 1. Avvio dell'applicazione**

L'applicazione MPI parte con un insieme iniziale di processi (P0...Pn). FlexMPI è attivo al suo interno.

**📊 1.a – Monitoraggio delle prestazioni**

Durante ogni iterazione, FlexMPI raccoglie le **metriche di prestazione** (come tempo di esecuzione, utilizzo CPU, efficienza, ecc.).

🧠 Questo è fatto dal modulo di **Performance Monitoring** ([M]), che poi restituisce il controllo all’applicazione (1.b).

**📉 2 – Valutazione delle prestazioni**

Alla fine dell’intervallo di campionamento, le metriche raccolte vengono inviate al modulo di **Performance Evaluation** ([MP, CPM]), che decide **se e come riconfigurare** l’applicazione.

**⚙️ 3 – Riconfigurazione dinamica**

A seconda del risultato della valutazione, FlexMPI può decidere:

**➕ 3.a – Espansione**

Se serve più potenza:

* Vengono **creati p nuovi processi** (Spawn p processes) e aggiornato il **communicator MPI**.
* Viene **ridistribuito il carico di lavoro e i dati** tra tutti i m + p processi attivi (Redistribute data).
* L’esecuzione riprende con **più processi** (4.a).

**➖ 3.b – Riduzione**

Se si vuole risparmiare risorse:

* I dati vengono prima **ridistribuiti** tra i processi rimanenti (Redistribute data).
* Poi **p processi vengono rimossi** e il communicator MPI viene aggiornato.
* L’esecuzione riprende con **meno processi** (m - p) (4.b).

**🔄 Il ciclo continua**

L'applicazione continua ad alternare fasi di computazione e monitoraggio. Ogni intervallo può portare a una nuova riconfigurazione, in base a:

* criteri di efficienza,
* obiettivi di performance,
* consumo energetico,
* politiche definite dall’utente o dal sistema.

**✅ In sintesi**

FlexMPI controlla il comportamento dell'applicazione MPI in base a ciò che osserva a runtime. Se rileva che può migliorare le prestazioni o risparmiare risorse, **aumenta o riduce** il numero di processi attivi, **senza fermare il programma**, adattandosi dinamicamente all’ambiente di esecuzione.

**🧠 Cos'è l’Intelligent Controller (IC)?**

L’**Intelligent Controller** è un componente centrale che **coordina e ottimizza l’uso delle risorse** nei sistemi paralleli su larga scala. Funziona come un **cervello distribuito**, che raccoglie informazioni da diverse fonti e prende decisioni **a runtime** per adattare il comportamento delle applicazioni e del sistema.

**🔗 Come comunica e con chi interagisce?**

L’IC **dialoga con tutti gli altri componenti** del sistema tramite **RPC** (Remote Procedure Call), cioè chiamate remote che gli permettono di scambiare dati e comandi con:

* Il **monitoring manager**: da cui riceve dati sulle prestazioni a runtime.
* Le **applicazioni parallele**: con cui coordina le operazioni di riconfigurazione.
* Gli **utenti**: da cui può ricevere preferenze o limiti da rispettare.
* Il **job scheduler** (es. **Slurm**): per avviare, fermare o riassegnare risorse.
* Il **malleability manager**: che applica concretamente i cambiamenti nella configurazione (es. aggiunta o rimozione di processi).
* L’**I/O scheduler**: per sincronizzare le operazioni di input/output.
* Il **sistema di storage**: per monitorare e ottimizzare l’accesso ai dati.

**🎯 Obiettivi principali**

L’obiettivo dell’IC è duplice:

**1. Costruire un’infrastruttura di controllo distribuita**

* Il sistema appare come una **“single system image”**: un ambiente unificato e coerente per l’utente.
* Usa **protocolli di coerenza distribuiti** per mantenere sincronizzati i dati tra tutti i nodi.

**2. Garantire un uso efficiente delle risorse**

* Sotto **carichi dinamici**, l’IC adatta il sistema in tempo reale.
* Migliora la **scalabilità** e riduce gli sprechi di risorse in ambienti complessi e condivisi (es. HPC, cloud).

**⚙️ Come opera nella malleabilità?**

Nel contesto di **FlexMPI**, è proprio l’**Intelligent Controller** che:

* Decide **quando applicare una riconfigurazione** (espansione o riduzione dei processi).
* **Invia i comandi malleabili** alla libreria FlexMPI.
* Coordina l’**Application Manager**, che si occupa di inviare i dati alla **regione malleabile** (cioè alla parte dell’applicazione che può essere adattata).

**🧩 In sintesi**

L’Intelligent Controller è una **componente fondamentale** per ottenere esecuzioni intelligenti e dinamiche nei sistemi HPC moderni. Fornisce una visione centralizzata e decisionale, ma opera in modo **distribuito**, **coerente** e **invisibile all’utente**.

**🔁 Cos’è l’RPC ?**

L'**RPC (Remote Procedure Call)** è un meccanismo che permette a un componente (es. un'applicazione MPI) di **comunicare con l’IC o ricevere comandi da esso**, anche se si trovano su nodi diversi. È come se una funzione venisse chiamata "a distanza", in un altro programma o macchina. Esistono **due tipi di protocolli**:

**➡️ 1. Comunicazione monodirezionale (componente → IC)**

Questo tipo è usato quando **un componente deve inviare informazioni all’IC**, ma **non si aspetta una risposta diretta**.

**Flusso:**

1. L’**IC registra le RPC generiche** e i callback necessari.
2. Il **componente** o l’applicazione **invia i dati chiamando una funzione RPC** (es. per segnalare l’inizio o la fine di una regione malleabile).
3. L’IC **memorizza l’ID e l’indirizzo** del componente che ha inviato i dati.
4. Il componente può **ripetere l’operazione quante volte serve**.
5. Ogni volta, l’**IC verifica se ci sono informazioni utili** da elaborare o archiviare.

💡 **Esempio:** un'app manda all’IC un segnale ogni volta che entra/esce da una regione malleabile.

**🔄 2. Comunicazione bidirezionale (IC ↔ componente)**

Usata quando è l’**IC a inviare comandi al componente**, che li **riceve e agisce di conseguenza**.

**Flusso:**

1. L’**IC crea i servizi RPC**, cioè le funzioni e callback da usare.
2. Il **componente registra la propria RPC** e **invia il proprio indirizzo** all’IC.
3. L’IC **memorizza l’ID e l’indirizzo** del componente.
4. L’applicazione esegue il suo codice normalmente.
5. L’IC, se necessario, **valuta azioni da intraprendere** (es. ridurre o aumentare i processi).
6. L’**IC invia i comandi** al componente.
7. Il **componente riceve e applica** le azioni richieste.

💡 **Esempio:** l’IC decide di espandere un’applicazione malleabile e invia un comando per aggiungere processi.

Nota: Il quarto file lo salto in quanto sono solo esempi, ergo passo direttamente al quinto file.

**05-FCAPP-Ad\_Hoc\_File\_Systems**

**🔸 Problema attuale nei cluster HPC**

Anche se i cluster moderni sono molto potenti, **la maggior parte dello storage è ancora basata su dischi magnetici tradizionali**, che sono:

* più **lenti**
* **meno adatti** a operazioni di I/O intensive tipiche di molte applicazioni scientifiche

Nel frattempo, sono disponibili **tecnologie di storage molto più veloci**:

* SSD (dischi a stato solido)
* NVRAM (memoria non volatile, ma più veloce dei dischi)

Queste tecnologie **sono presenti nei nodi di calcolo**, ma **non vengono quasi mai sfruttate automaticamente** nei flussi di lavoro scientifici.

**🔸 Perché non si usano i nuovi dispositivi di storage?**

* **Integrazione difficile**: l’uso di SSD o NVRAM richiede spesso **configurazioni manuali** complicate
* **Gli utenti non hanno tempo o competenze** per modificarle
* Risultato: anche se **lo storage veloce è disponibile**, **viene ignorato** e si usano ancora i dischi lenti

**✅ Soluzione: Ad Hoc File Systems**

L’idea è **creare file system temporanei (ad hoc)** direttamente sui nodi del cluster, per:

* **Usare in automatico lo storage locale veloce (SSD/NVRAM)**
* **Migliorare drasticamente l’I/O** delle applicazioni scientifiche
* Usarli per **una singola applicazione o un’intera campagna** di simulazioni

Questi ad hoc file systems possono essere:

* **Effimeri**: si creano all’inizio del job e si distruggono alla fine
* **Ottimizzati**: progettati solo per quell’applicazione, quindi massimizzano le prestazioni
* **Integrabili** con strumenti di scheduling (es. SLURM) e monitoraggio

**🔧 Cos’è FUSE?**

**FUSE (Filesystem in Userspace)** è un’interfaccia software per sistemi operativi Unix e simili (come Linux, macOS, Windows, BSD…) che consente **agli utenti non privilegiati di creare i propri file system**, **senza modificare il kernel**.

In pratica, il codice che definisce il comportamento del file system non gira all’interno del kernel (come nei file system tradizionali), ma nello **spazio utente**. Questo rende **lo sviluppo più semplice e sicuro**, perché eventuali errori non compromettono il sistema operativo.

Il modulo FUSE nel kernel si occupa di fare da **ponte** tra:

* le richieste del sistema (come read(), write(), stat()) → dal kernel
* e la tua applicazione handler → nello spazio utente

Immagine che contiene testo, schermata, diagramma, Piano

Il contenuto generato dall'IA potrebbe non essere corretto.

Immagine che contiene testo, Carattere, linea, schermata

Il contenuto generato dall'IA potrebbe non essere corretto.

Immagine che contiene testo, schermata, Carattere, numero

Il contenuto generato dall'IA potrebbe non essere corretto.

**🧠 Cosa rende FUSE speciale?**

* Tutta la logica del file system può stare **nello spazio utente**, quindi:
  + Non serve modificare il kernel (niente codice kernel!).
  + È più sicuro: un errore non manda in crash il sistema.
  + È portabile su più sistemi operativi.
  + È facile da sviluppare e testare.

**⚙️ Come funziona FUSE?**

1. Scrivi un programma che descrive **come il tuo file system deve rispondere alle richieste**.
2. Lo **colleghi alla libreria libfuse** e lo lanci.
3. Al **montaggio del file system**, questo programma viene **registrato dal kernel**.
4. Quando un utente fa una richiesta (es. ls, cat, cp), il kernel la **inoltra al tuo programma**, che genera la risposta.

Questo meccanismo è perfetto per implementare **file system virtuali**: sistemi che non salvano fisicamente dati, ma agiscono come un’interfaccia tra l’utente e una fonte dati (un archivio, un backup, una risorsa remota...).

**🧱 Applicazioni di FUSE**

FUSE è estremamente versatile e ha tantissime applicazioni:

**📀 File system tradizionali on-disk**

FUSE può essere usato anche per implementare file system veri e propri su disco:

* **NTFS-3G**: per accedere a partizioni Windows NTFS da Linux
* **Linear Tape File System**: per trattare dati su nastro come se fossero su disco
* **retro-fuse**: per montare vecchi file system UNIX (5a–7a edizione, BSD)

**🧰 File system virtuali personalizzati**

Con FUSE puoi creare **file system ad hoc**, che possono:

* Avere un backend su **RAM, SSD, cloud, database, rete…**
* Offrire un’interfaccia trasparente: l’utente interagisce come con un file system normale, ma l’implementazione può fare qualunque cosa

**💾 File system remoti e per HPC**

È proprio in questo ambito che **FUSE mostra il suo potenziale nei sistemi ad alte prestazioni (HPC)**:

* **BeeOND, GekkoFS, BurstFS** sono file system temporanei/veloci usati per campagne scientifiche
* Sono costruiti con FUSE per usare SSD o NVRAM locali, **migliorando enormemente le prestazioni I/O**

**📂 File system per archiviazione e backup**

FUSE è usato anche per esplorare archivi compressi o backup **senza estrarli**:

* **archivemount**
* **Borg**, **Restic**, **SPFS**
* **Atlas (Rubrik)** per backup enterprise

**🚀 Perché è importante per HPC?**

Oggi molti cluster scientifici usano ancora dischi magnetici tradizionali. FUSE permette di **integrare tecnologie più rapide (come SSD/NVRAM)** direttamente nelle pipeline scientifiche, rendendo possibile:

* l’uso temporaneo e trasparente di storage locale veloce
* il miglioramento delle prestazioni delle applicazioni
* la costruzione di file system temporanei "ad hoc" su nodi di calcolo

In sintesi, **FUSE è la chiave per creare file system su misura**, dinamici, portabili e potenti, senza dover modificare il kernel del sistema operativo.

**🧩 Applicazioni di FUSE nei File System**

FUSE, grazie alla sua flessibilità e al fatto che funziona nello spazio utente, è usato **non solo per sviluppare file system virtuali**, ma anche per **ricreare file system reali** o adattarne di nuovi per esigenze moderne. Ecco dove e come viene applicato:

**📀 File System On-Disk Convenzionali**

FUSE può essere usato per implementare **file system classici**, anche quelli basati su dischi fisici. Questo approccio può essere utile in caso di:

* **vincoli di licenza** (es. software proprietario)
* **compatibilità** con vecchi sistemi o ambienti

Esempi:

* **Linear Tape File System**: consente di accedere ai file salvati su **nastri magnetici** (tecnologia ancora molto usata in ambienti scientifici e archivi a lungo termine) **come se fossero su disco**, con comandi standard tipo ls o cp.
* **NTFS-3G** e **Captive NTFS**: permettono l’accesso in lettura/scrittura alle partizioni **NTFS** (Windows) da sistemi Linux.
* **Retro-fuse**: rende possibile montare file system **creati su vecchi sistemi Unix** (come 5a–7a edizione di UNIX, BSD), su OS moderni.

**🌐 File System Remoti e Distribuiti**

FUSE è anche un’ottima scelta per implementare **client di file system remoti o distribuiti**, cioè per accedere a risorse di archiviazione che non risiedono localmente sul computer. In particolare è fondamentale nei sistemi HPC (High Performance Computing) moderni.

**💡 Ad-Hoc File System per HPC**

Un utilizzo avanzato di FUSE è la creazione di **file system temporanei e ad hoc** su sistemi HPC per migliorare le prestazioni delle applicazioni scientifiche. Questi file system usano storage veloce locale (SSD o NVRAM) e sono progettati per essere montati e smontati dinamicamente, a seconda del carico.

Esempi pratici:

* **BeeOND**: crea un file system temporaneo BeeGFS su un sottoinsieme di nodi HPC, migliorando l'I/O durante l'esecuzione di un'applicazione.
* **GekkoFS**: file system leggero, distribuito, ottimizzato per **applicazioni scientifiche con accessi irregolari ai dati**.
* **BurstFS**: pensato per fornire un layer di burst buffer (memoria temporanea veloce) per sistemi paralleli.

**🗂️ Accesso Diretto a Backup e Archivi**

Un’altra categoria molto utile di applicazioni FUSE è l’**esplorazione diretta di backup e archivi compressi** senza doverli estrarre:

* **archivemount**: consente di "montare" archivi compressi (.tar.gz, .zip) come se fossero directory normali.
* **Atlas (Rubrik)**: file system distribuito e immutabile usato per la protezione dei dati in ambito enterprise.
* **Borg**: software di backup che **deduplica** i dati e consente di esplorare i backup montandoli con FUSE.
* **Restic**: soluzione moderna di backup, usa FUSE per rendere ogni **snapshot** esplorabile come una cartella.
* **SPFS**: pensato per **Spectrum Protect**, permette di montare lo spazio dei backup come un normale file system, sfruttando funzionalità come compressione, deduplicazione, cifratura. È un file system WORM (Write Once, Read Many).

**✅ Conclusione**

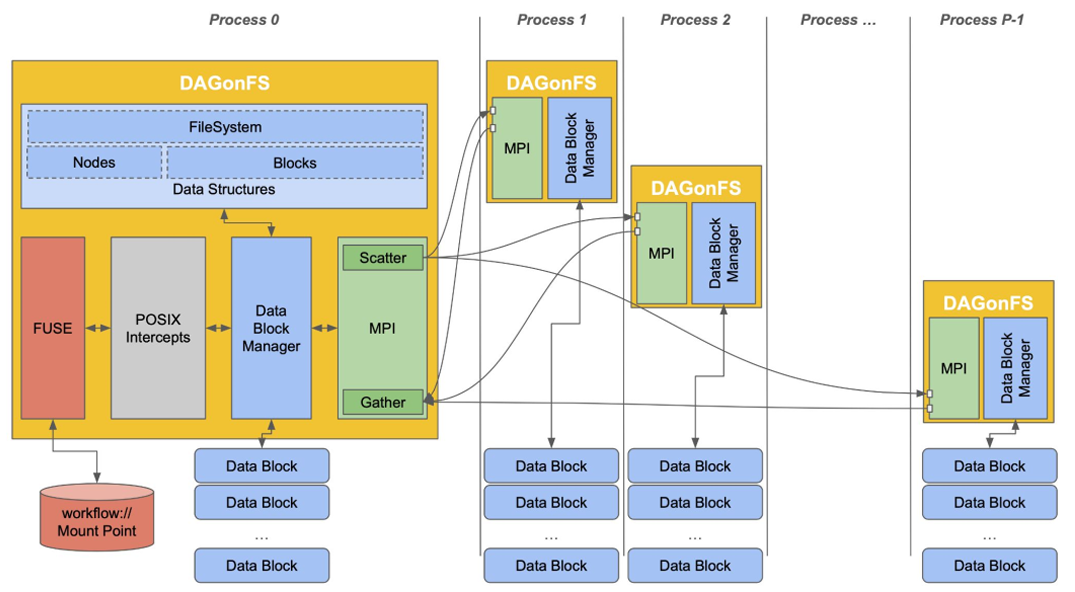
FUSE non è solo uno strumento per sviluppatori, ma una vera piattaforma per costruire **file system flessibili, portabili e potenti**, che vanno **dai backup alla ricerca scientifica**. Permette agli sviluppatori di:

* sfruttare tecnologie nuove (SSD, NVRAM)
* creare file system ad hoc per applicazioni temporanee
* accedere a dati legacy o remoti con la stessa semplicità dei file locali

**🧩 Cos'è DAGonFS?**

**DAGonFS** è un **file system ad-hoc** pensato per essere usato in combinazione con **workflow engine**, cioè quei sistemi che gestiscono e coordinano flussi di lavoro complessi (come DAGonStar, un motore di workflow citato nel testo).

In parole semplici, DAGonFS è un **file system temporaneo e distribuito** progettato per supportare l’esecuzione efficiente di workflow scientifici (es. simulazioni, analisi dati HPC) direttamente nella memoria dei nodi di calcolo.



**🧠 Componenti chiave di DAGonFS**

1. **FUSE (Filesystem in Userspace)**  
   Come abbiamo visto prima, FUSE permette di creare file system personalizzati nello spazio utente. DAGonFS lo usa per:
   * Montare un file system virtuale.
   * Gestire l’interfaccia file tra applicazione e dati.
   * Evitare di dover modificare il kernel o usare file system complessi sul disco.
2. **RAM (memoria volatile)**  
   I dati non vengono scritti su disco, ma **tenuti in RAM**, rendendo le operazioni di I/O molto più **veloci**. È ideale per file temporanei e accessi intensivi.
3. **MPI (Message Passing Interface)**  
   Per garantire che il file system sia **scalabile** e **distribuito** su più nodi, DAGonFS usa **MPI** per:
   * Coordinare i dati tra diversi nodi.
   * Comunicare tra le varie istanze del file system montato in RAM.

**🧪 A cosa serve DAGonFS?**

DAGonFS è utile nei casi in cui:

* Hai **workflow scientifici** (molteplici task in sequenza o in parallelo).
* Hai bisogno di prestazioni I/O elevate (es. scrittura/lettura veloce dei file).
* Non puoi o non vuoi usare un file system permanente su disco.
* Lavori in un ambiente **HPC** (High Performance Computing) e vuoi sfruttare al massimo le **risorse locali** dei nodi (RAM + CPU).

**🔍 Cos'è ADMIRE?**

**ADMIRE** è un progetto pensato per rispondere alle nuove sfide poste dall’elaborazione di enormi volumi di dati nei sistemi **HPC** (High Performance Computing), in particolare in vista dell’**exascale computing**.

Il problema di fondo è che i **file system tradizionali**, centralizzati e piatti, **non sono più sufficienti**: l’accesso ai dati non coordinato e la **banda limitata** generano colli di bottiglia significativi. Inoltre, i moderni sistemi HPC includono **storage multi-tier** (RAM, SSD, NVRAM…), ma **manca un controllo intelligente** per gestirli in modo efficace.

**🎯 Obiettivi di ADMIRE**

ADMIRE propone una soluzione software-defined per **creare uno stack I/O dinamico e intelligente**, in grado di:

* **Adattarsi alle esigenze I/O dell'applicazione** in ogni fase del suo ciclo di vita.
* **Controllare e coordinare** l’uso delle risorse computazionali e di storage.
* **Estendere o ridurre dinamicamente i file system ad hoc**, in base alla richiesta di I/O.
* **Bilanciare le prestazioni tra calcolo e I/O**, migliorando il throughput e l'efficienza.

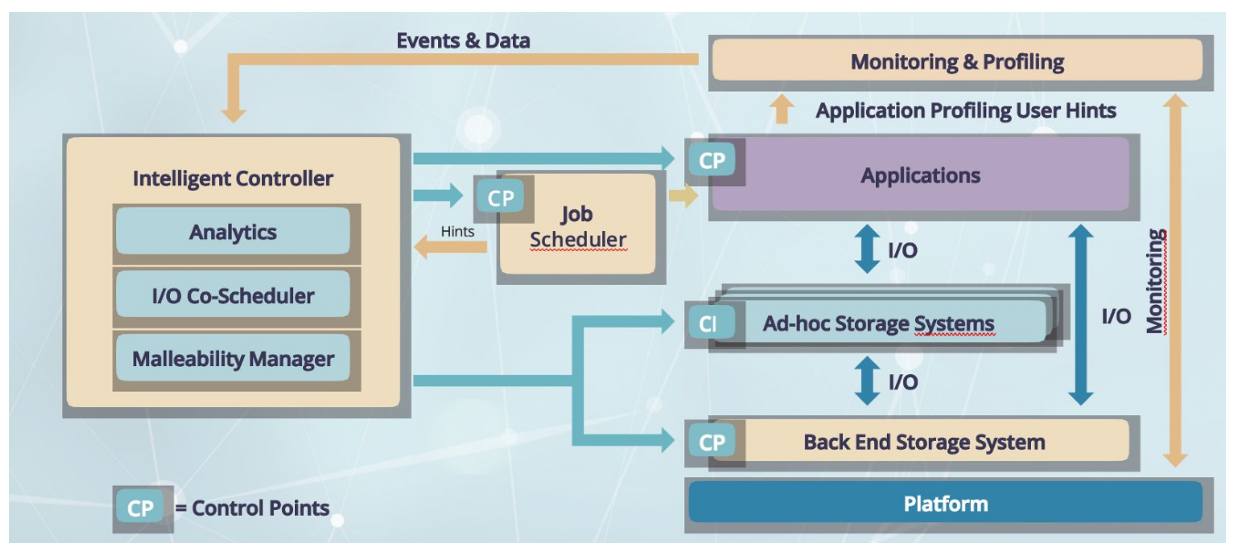
Questo è possibile grazie all’**integrazione della malleabilità**, ovvero la capacità di adattare **dinamicamente** risorse e configurazioni di calcolo e archiviazione.

**🧠 Come funziona ADMIRE?**

ADMIRE si basa su un’infrastruttura **software-defined** che separa:

* Il **percorso dei dati** (cioè il flusso effettivo delle informazioni).
* Il **percorso di controllo**, che governa come e dove avviene l’I/O, attraverso **monitoraggio, coordinamento e orchestrazione** intelligente.

Tutto questo è fatto in modo **scalabile**, grazie a un controller che monitora e regola le prestazioni in tempo reale, e prende decisioni per ottimizzare l'esecuzione del workflow.



**💾 Sistemi di storage ad-hoc supportati da ADMIRE**

ADMIRE implementa **file system temporanei e ad-hoc**, ottimizzati per velocità, resilienza e scalabilità. Questi sono costruiti per sfruttare tecnologie come:

* **RAM** (velocissima, volatile)
* **SSD/NVM** (non volatili, veloci)
* Altri supporti di archiviazione locale

I principali **file system ad-hoc** usati in ADMIRE sono:

* **GekkoFS**: file system distribuito in spazio utente.
* **Hercules IMSS**: specializzato per I/O intermedio in RAM.
* **Expand**: permette l’espansione dinamica dello storage.
* **dataClay**: orientato agli oggetti (object-oriented data store), ideale per workflow che trattano strutture complesse.

➡️ Tutti questi sistemi supportano API **POSIX** (standard per l’I/O nei sistemi Unix), permettendo così di essere compatibili con la maggior parte delle applicazioni. Tuttavia, sono possibili anche interfacce I/O **personalizzate**, se l’applicazione è in grado di supportarle.

**🐾 GekkoFS: Cos’è e perché è utile**

**GekkoFS** è un **file system distribuito in spazio utente** (user-space) progettato specificamente per ambienti **HPC** (High Performance Computing). La sua missione è **sfruttare l'I/O locale** dei nodi di calcolo (es. SSD, RAM) per creare uno **storage temporaneo ad alte prestazioni**, isolato dal file system centrale del cluster.

➡️ **Cosa significa?** Ogni job può usare uno spazio di archiviazione indipendente, **senza interferire con altri job**, migliorando **le prestazioni I/O** e **l’isolamento**.

Immagine che contiene testo, schermata, design

Il contenuto generato dall'IA potrebbe non essere corretto.

**🧠 Come funziona GekkoFS**

GekkoFS è **interamente in user space**, cioè non richiede modifiche al kernel del sistema operativo. Questo lo rende **facile da installare ed eseguire** anche senza privilegi da amministratore.

È composto da **due elementi principali**:

1. **Client library (interposizione)**
   * L’applicazione si collega a questa libreria tramite LD\_PRELOAD (una variabile d’ambiente di Linux).
   * La libreria **intercetta** tutte le chiamate I/O (open, read, write, ecc.).
   * Se l’I/O è destinato a un file all’interno del mount point virtuale di GekkoFS, la chiamata viene **gestita da GekkoFS**.
   * Altrimenti, l’I/O passa al file system del sistema operativo.
2. **Server/Daemon**
   * È il processo che **gestisce effettivamente i dati**, aggregando lo storage locale del nodo (RAM, SSD, ecc.).
   * Ogni nodo del cluster esegue il proprio daemon.
   * I dati possono essere replicati o distribuiti tra più nodi, rendendo il file system **scalabile e resiliente**.

**🕒 Un file system temporaneo ed effimero**

A differenza di file system permanenti come Lustre o GPFS, **GekkoFS è effimero**:

* Viene **creato dinamicamente** all’inizio del job HPC.
* Viene **terminato alla fine del job**.
* I dati **non persistono**, quindi è **necessario copiare i file importanti** su un file system permanente se si vogliono conservare.

➡️ Questo approccio rende GekkoFS perfetto per:

* Simulazioni e analisi temporanee,
* Fasi con accesso I/O molto intensivo,
* Workflow che beneficiano di storage veloce locale senza congestionare il backend.

**🧩 Perché è adatto ad ADMIRE e all’HPC moderno**

Nel contesto del framework **ADMIRE**, GekkoFS è uno degli ad-hoc file system usati per:

* Fornire **I/O on-demand ad alte prestazioni**,
* Supportare la **malleabilità I/O**, cioè la capacità di **espandere o ridurre lo storage temporaneo** in base al carico dell’applicazione,
* Ridurre la dipendenza da file system centralizzati, evitando **collo di bottiglia** su larga scala.

**📌 Conclusione**

GekkoFS rappresenta un modo moderno, flessibile e scalabile per gestire i dati **durante l’esecuzione dei job HPC**, offrendo alte prestazioni e isolamento. È una soluzione **utente-centrica**, pensata per **ottimizzare l’I/O locale temporaneo**, e perfettamente in linea con l’obiettivo di **ADMIRE** di controllare dinamicamente le risorse nei sistemi HPC.

**📦 Cos'è dataClay?**

**dataClay** è un sistema di archiviazione **object-oriented distribuito**, progettato per essere **flessibile** e **intelligente** nella gestione dei dati in ambienti ad alte prestazioni (HPC) o altamente distribuiti (come edge-to-cloud). A differenza di file system classici, dataClay **non si limita a memorizzare dati**, ma fornisce anche **funzionalità di esecuzione associate agli oggetti stessi**.

**🔍 Caratteristiche chiave**

* **Oggetti intelligenti**: ogni oggetto può contenere **dati e metodi**, cioè anche codice utente. Questo approccio riduce la necessità di comunicazioni costose tra nodi, poiché parte della logica è **eseguita localmente** vicino ai dati.
* **Trasparenza della distribuzione**: dataClay **nasconde i dettagli** di dove sono fisicamente memorizzati gli oggetti. L’utente non deve preoccuparsi della collocazione: lavora con oggetti come se fossero locali, anche se distribuiti.
* **Gestione in memoria**: gli oggetti possono essere **gestiti direttamente in RAM**, evitando conversioni e accessi frequenti a disco. Questo migliora notevolmente le **prestazioni I/O**.
* **Modalità di utilizzo**:
  + Come **servizio permanente** (simile a un database distribuito).
  + Come **sistema effimero**, cioè temporaneo, utile in ambienti HPC per sessioni di lavoro isolate.

**🧱 Architettura**

La struttura di dataClay è divisa in **due componenti principali**:

**1. Logic Module**

È il **cuore centrale di controllo**:

* **Metadata Manager**: tiene traccia della posizione degli oggetti, assegnando a ciascuno un ID unico interno.
* **Class Manager**: gestisce la **struttura semantica** degli oggetti (tipo, attributi, metodi, ecc.).
* **Session Manager**: coordina le **sessioni degli utenti** che accedono al sistema.

**2. Backends**

Ogni backend contiene:

* **Storage Location**: la parte dove gli oggetti vengono fisicamente **memorizzati**.
* **Execution Environment**: un **motore locale** che riceve ed esegue le richieste fatte sugli oggetti gestiti da quel backend (ad esempio chiamate a metodi sugli oggetti).

**⚙️ Perché è utile in HPC (es. ADMIRE)?**

Nel contesto di **ADMIRE**, dataClay:

* Fornisce **archiviazione intelligente e adattiva** per applicazioni che devono elaborare grandi quantità di dati.
* Permette un **uso efficiente delle risorse** e un’ottima **scalabilità**, specialmente quando i dati devono “seguire” il calcolo o adattarsi al carico.
* Essendo user-level, può essere integrato facilmente con applicazioni scientifiche, **senza privilegi amministrativi**.

**✅ In sintesi**

dataClay non è solo uno spazio per memorizzare dati, ma un **data store orientato agli oggetti** che migliora la **gestione, la località e la performance** in ambienti distribuiti e HPC. È uno strumento **chiave** per superare i limiti degli stack I/O tradizionali e implementare **sistemi di archiviazione intelligenti e flessibili** come richiesto dalle moderne architetture exascale.

**🧠 Cos’è Hercules?**

**Hercules** è un **acceleratore I/O in-memory** progettato per gestire dati volatili e ottimizzare l’I/O nei workflow HPC. Fornisce un **file system effimero, temporaneo**, basato in RAM, con interfaccia **compatibile POSIX** per garantire **portabilità** e semplicità d’uso. È pensato per supportare applicazioni scientifiche ad alte prestazioni che richiedono throughput elevato, come checkpointing, distribuzione dei dati e gestione concorrente.

**⚙️ Architettura generale**

L’architettura di Hercules si basa su due **layer principali**:

**🔸 Client Layer**

* Ogni applicazione utilizza una **libreria client IMSS** (In-Memory Storage System).
* I client gestiscono **istanze IMSS**, ciascuna identificata da un URI, per creare, accedere e modificare dati in memoria.
* Un’istanza IMSS è una **unità di storage temporanea distribuita** tra più nodi, che usa la **RAM** per memorizzare dataset.

**🔸 Back-end Layer**

* Ogni istanza IMSS ha **diversi server distribuiti** che conservano i blocchi di dati.
* Ogni server ha un **dispatcher thread**, che smista le richieste tra **worker thread** usando round-robin.
* I worker mappano ogni blocco a una posizione in memoria tramite una struttura dati a mappa.

**🔄 Evoluzione: da ZeroMQ a UCX**

Hercules è passato da **ZeroMQ** al più moderno **UCX (Unified Communication X)** per gestire la comunicazione tra client e server. UCX è pensato per reti ad alta velocità e bassa latenza (come **InfiniBand**, **RoCE**, **TCP/IP**) ed espone primitive astratte molto performanti, tra cui:

* RDMA,
* accesso diretto GPU,
* comunicazione asincrona,
* copia-zero (zero-copy),
* isolamenti QoS,
* trasferimenti two-sided.

**➕ Vantaggi di UCX**

* Alta **scalabilità**.
* Trasmissione ottimizzata grazie a **zero-copy**.
* Supporto a più **protocolli e interfacce di rete**.
* Comunicazioni **peer-to-peer asincrone** con **connessioni isolate** per dati e metadati.

Immagine che contiene testo, schermata, Carattere, numero

Il contenuto generato dall'IA potrebbe non essere corretto.

**🔁 Malleabilità e flessibilità**

Hercules è **malleabile**: può **espandere o ridurre dinamicamente il numero di nodi di storage**, in base alle esigenze I/O dell'applicazione. Questo permette di:

* aumentare il throughput durante fasi ad alto I/O,
* ridurre risorse quando non servono, risparmiando memoria.

**📊 Politiche di distribuzione dei dati**

Hercules supporta **diverse strategie** per distribuire i blocchi dati tra i server:

* **ROUND ROBIN**: distribuzione uniforme sequenziale.
* **BUCKETS**: i dati vengono suddivisi in blocchi consecutivi assegnati a ciascun server.
* **HASHED**: si usa una funzione hash sulla chiave per decidere la destinazione.
* **CRC16/CRC64**: simili a hash ma usando checksum a 16 o 64 bit.
* **LOCALE**: il blocco viene gestito dal server nello stesso nodo del client.

**🧩 Strategie di esecuzione**

Hercules può essere configurato in due modalità operative:

* **Application-attached**: ogni nodo dell’applicazione esegue sia il **client** che il **server IMSS**, condividendo risorse localmente. Questo migliora la **co-localizzazione** dei dati.
* **Application-detached**: i **client** sono nei nodi applicativi e i **server** sono distribuiti **fuori dal perimetro dell’applicazione** (es. in nodi offshore). Si sfrutta così tutta la potenza computazionale disponibile senza saturare i nodi di calcolo.

**✅ In sintesi**

Hercules è un sistema di storage in-memory, **effimero, distribuito e malleabile**, progettato per ambienti HPC e capace di:

* adattarsi alle necessità dell’applicazione,
* sfruttare al meglio le risorse di rete e memoria,
* garantire portabilità e compatibilità POSIX.

**🔄 Expand – File System ad-hoc parallelo e distribuito**

**Expand** è uno dei file system temporanei e flessibili usati nel progetto **ADMIRE** per migliorare la gestione dell’I/O nei sistemi HPC. A differenza di soluzioni in-memory come **Hercules** o orientate agli oggetti come **dataClay**, Expand si basa su **storage locali fisici** (come SSD o HDD) ma con una struttura distribuita e parallela. È particolarmente utile in ambienti dove è importante **sfruttare lo storage disponibile sui nodi di calcolo**.

**📦 Architettura e funzionamento**

* **Struttura distribuita**: Expand è composto da diversi **server storage** che operano **sui nodi di calcolo stessi**. Questi server comunicano tra loro usando **MPI**, per assicurare **scalabilità e coordinamento** efficiente.
* **Archiviazione locale**: Ogni nodo utilizza i **dischi locali (HDD o SSD)** tramite le funzioni standard del sistema operativo. Non serve hardware speciale né supporto del kernel.
* **Partizione parallela**: I vari server MPI combinano le loro directory locali per costruire una **partizione parallela distribuita**. Questa partizione appare come un unico file system virtuale per le applicazioni utente.
* **Distribuzione dei file**:
  + Ogni file è suddiviso in **sottofile**, distribuiti su più server.
  + I file possono essere **strippati** (striped): cioè i dati sono distribuiti a blocchi su tutti i server secondo uno schema **round-robin** (ciclico), facilitando l’**accesso parallelo** e migliorando le prestazioni.
  + Gli utenti non vedono i sottofile: la divisione è **trasparente**, come se lavorassero su un file system classico.

Immagine che contiene testo, schermata, diagramma, linea

Il contenuto generato dall'IA potrebbe non essere corretto.

**🔧 Integrazione con ADMIRE**

Nel contesto **ADMIRE**, Expand fa parte dello **stack di storage ad-hoc temporanei**, usati per:

* **Adattare dinamicamente lo storage** in base al comportamento I/O dell’applicazione.
* **Ridurre il carico sul file system centrale** (back-end).
* **Aumentare il throughput** grazie all'accesso parallelo e distribuito.
* **Supportare malleabilità**, ampliando o riducendo la dimensione del file system secondo necessità.

**✅ Quando usare Expand**

Expand è una scelta ideale quando:

* Si vuole **ottimizzare l’I/O su dischi locali**, senza caricare il file system centrale.
* Si gestiscono **file di grandi dimensioni** che beneficiano dello striping.
* Si richiede **scalabilità**, ma senza ricorrere a RAM o object store.
* Serve un file system **temporaneo**, per la durata di una singola esecuzione.