Alma Mater Studiorum · Università di Bologna

Campus di Cesena

SCUOLA DI INGEGNERIA E ARCHITETTURA

Corso di Laurea in Ingegneria e scienze informatiche

ANALISI DELLE PRESTAZIONI DEL CLUSTER RISC-V: MONTE CIMONE

Relatore:
Chiar.mo Prof.
MORENO MARZOLLA

Presentata da:
GIOVANNI
ANTONIONI

II Sessione di laurea
2021 / 2022

Dedicato con affetto alla mia famiglia e ai miei amici
che mi hanno accompagnato in questo percorso.

Sommario

Gli sforzi di ricerca relativi all'High Performance Computing, nel corso degli anni, hanno prodotto risultati importanti inerenti all'incremento delle prestazioni sia in termini di numero di operazioni effettuate per periodo temporale, sia introducendo o migliorando algoritmi paralleli presenti in letteratura.

Tali traguardi hanno comportato cambiamenti alla struttura interna delle macchine; si è assistito infatti ad un'evoluzione delle architetture dei processori utilizzati e all'impiego di GPU come risorse di calcolo aggiuntive.

La conseguenza di un continuo incremento di prestazioni è quella di dover far fronte ad un grosso dispendio energetico, in quanto le macchine impiegate nell'HPC sono ideate per effettuare un'intensa attività di calcolo in un periodo di tempo molto prolungato; l'energia necessaria per alimentare ciascun nodo e dissipare il calore generato comporta costi elevati.

Tra le varie soluzioni proposte per limitare il consumo di energia, quella che ha riscosso maggior interesse, sia a livello di studio che di mercato, è stata l'integrazione di CPU di tipologia RISC (Reduced Instruction Set Computer), in quanto capaci di ottenere prestazioni soddisfacenti con un impiego energetico inferiore rispetto alle CPU CISC (Complex Instruction Set Computer).

In questa tesi è presentata l'analisi delle prestazioni di Monte Cimone, un cluster composto da 8 nodi di calcolo basati su architettura RISC-V e distribuiti in 4 piattaforme (*blade*) dual-board. Verranno eseguiti dei benchmark che ci permetteranno di valutare:

- le prestazioni dello scambio di dati a lunga e corta distanza;
- le prestazioni nella risoluzione di problemi che presentano un principio di località spaziale ridotto;
- le prestazioni nella risoluzione di problemi su grafi e, nello specifico, ricerca in ampiezza e cammini minimi da sorgente singola.

Indice

Elenco delle figure

Capitolo 1

Concetti di HPC e architetture parallele

Con il termine High Performance Computing (HPC) si fa riferimento all'insieme delle tecniche, soluzioni hardware e applicazioni software che permettono di creare un sistema di elaborazione composto da più calcolatori, in grado di eseguire computazioni onerose nel minor tempo possibile ricorrendo ad un modello di programmazione parallelo.

1.1 Evoluzione dei processori

L'aumento nel corso degli anni della densità dei transistor all'interno di un processore trova conferma in quanto enunciato da Gordon E. Moore nel 1965 con quella che fu definita "Legge di Moore" (*Moore's Law*) [16], dove si sosteneva che il numero di transistor all'interno di un circuito integrato sarebbe raddoppiato ogni due anni. Tale andamento tese a diminuire tuttavia nel momento in cui iniziarono ad emergere due problematiche fondamentali realative a:

- dimensione minima che possono assumere i transistor. Nei processori moderni si stima che la grandezza di questi, possa variare dai $10nm$ ai

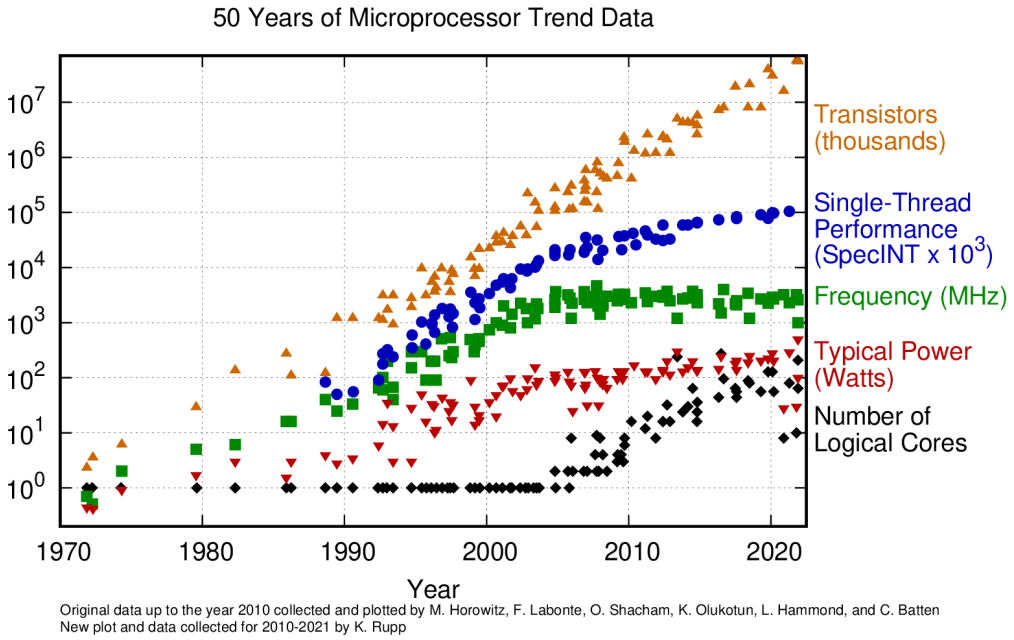


Figura 1.1: Evoluzione delle caratteristiche dei processori

$20nm$ e che recentemente si sia riusciti a creare delle porte logiche di dimensione di $1nm$ [8].

- quantità di calore creata all'interno di un processore: un elevato numero di transistor permette di avere delle prestazioni elevate ma un conseguente aumento dell'impiego energetico. L'energia adoperata viene convertita in una quantità di calore difficile da dissipare e tale da rendere instabile il processore.

Nella figura 1.1 viene mostrata l'evoluzione dei processori dal 1970 al 2021. Come è possibile notare, vi è dall'inizio degli anni 2000 una stabilizzazione della frequenza media (in GHz) ed un incremento del numero di core logici a partire dal 2005. Il miglioramento delle prestazioni non è più dato solamente dall'aumento della frequenza del clock, ma anche dal numero di unità operazionali alle quali è possibile distribuire il lavoro.

Il concetto di suddivisione del lavoro può essere esteso non solo ai core ma anche a più calcolatori, mettendo questi in comunicazione tra loro.

1.2 Estensione dell'architettura di Von Neumann

L'architettura di Von Neumann descrive una struttura di calcolatore astratta basata su tre componenti principali: un'unità centrale (CPU), il cui compito è eseguire una serie di operazioni su un insieme di dati, una memoria, dove vengono scritte o lette delle informazioni ed un insieme di canali (bus) che permettono la comunicazione tra le due.

Le architetture parallele, ed in generale la totalità dei calcolatori moderni, basano il loro funzionamento sul modello descritto da Von Neumann, il quale presenta tuttavia un collo di bottiglia: la suddivisione tra area di calcolo e memoria fa si che le prestazioni siano influenzate dalla larghezza di banda dei canali di comunicazione che ne permettono l'interscambio dei dati.

Occorre adoperare dunque una serie di tecniche che possano migliorare tale modello al fine di aumentarne le prestazioni.

1.2.1 Caching

Per risolvere un problema legato fortemente all'accesso ai dati in memoria, è possibile agire anzitutto sulle metodologie usate per reperirli.

L'approccio più comune è quello del *caching*, che consiste nell'adoperare una gerarchia di memorie (*cache*) di dimensione e velocità differente, nelle quali vengono conservati i dati necessari alla CPU per l'esecuzione di istruzioni.

Normalmente cache piccole e veloci sono poste in prossimità del processore in modo da favorirne un accesso più rapido ai dati, mentre quelle più capienti, ma con tempo di accesso più lento, sono posizionate vicino alla memoria principiale.

Nel momento in cui il processore tenta di accedere a informazioni non presenti in cache (cache miss), vengono esaminate le varie memorie nell'ordine in cui queste risiedono nella gerarchia. Le cache verranno successivamente aggiornate con nuovi dati prelevati da posizioni contigue nella memoria (cache lines).

Se pur efficace in molti contesti, l'uso delle cache risulta meno efficiente per le applicazioni che esibiscono scarsa località temporale e/o spaziale. È sorta recentemente la necessità di dover ottimizzare una classe di applicazioni definita "memory-intensive" (o data intensive), la quale è caratterizzata da un periodico reperimento di dati in posizioni irregolari nella memoria. Una soluzione al seguente problema può essere l'utilizzo di un approccio PIM (Process In Memory), nel quale si tenta di risolvere la latenza introdotta dal collo di bottiglia dell'architettura di Von Neumann, eseguendo le operazioni direttamente nella memoria principale [37].

1.2.2 Instruction Level Parallelism

L'instruction-level parallelism (ILP) è la modalità tramite la quale si incrementano le prestazioni del processore facendogli eseguire più istruzioni contemporaneamente. L'esecuzione di un'istruzione avviene in cinque fasi distinte, così ordinate:

- recupero dell'istruzione dalla memoria (IF);
- decodifica dell'istruzione (ID);
- esecuzione (EX);
- attivazione della memoria (MEM);
- scrittura del risultato in un registro (WB).

Ciascuna di queste può essere eseguita da componenti interne al processore, denominate unità funzionali. L'ILP permette di utilizzarle tramite due tecniche: il *pipelining* ed il *multiple issue*. Il funzionamento del pipelining è

simile a quello di una catena di montaggio: ad ogni ciclo di clock, un insieme di unità funzionali collegate serialmente (*pipeline*) ed in grado di svolgere ciascuna delle fasi descritte in precedenza, esegue un compito specifico passando il suo output all'unità successiva. Quando una pipeline è a pieno regime, tutte le 5 fasi necessarie all'esecuzione dell'istruzione sono svolte in un unico ciclo di clock. Un processore può avere una o più pipeline al suo interno. Nel multiple issue, sono presenti all'interno della cpu più unità funzionali in grado di svolgere la stessa fase. Queste sono utilizzate in maniera indipendente per eseguire parallelamente le istruzioni di un programma. I processori che adoperano questa metodologia di ILP vengono detti superscalari. Si parla di multiple issue statico quando le unità funzionali che devono essere attivate ed usate sono decise in fase di compilazione del programma, mentre si parla di multiple issue dinamico quando le unità funzionali sono attivate ed adoperate in fase di esecuzione.

1.3 Tipologie di parallelismo

Esistono varie tecniche per effettuare delle operazioni parallele all'interno di calcolatori. Quanto descritto nel paragrafo 1.2.2 ha mostrato come sia possibile sfruttare più unità funzionali di un processore per eseguire in parallelo delle istruzioni. Vi sono tuttavia altri due importanti meccanismi che possono essere applicati: il *thread-level* e *vector parallelism*.

1.3.1 Thread-level parallelism

Il thread-level parallelism (TLP) è una tecnica per la quale un insieme di istruzioni di un programma viene eseguita in maniera parallela attraverso l'utilizzo di *thread*, che sono flussi indipendenti del programma dotati di un proprio *program counter* ed *instruction pointer*. Il parallelismo è dato sia dalla possibilità di mantenere attivi più thread contemporaneamente (in tal caso si parla di multithreading simultaneo), sia di poter alternare l'esecuzione di questi in base al carico di lavoro che devono svolgere. Come esempio

per quest'ultimo caso, si consideri la situazione nella quale un determinato thread debba attendere il caricamento di dati dalla memoria: nell'attesa che l'operazione venga completata è possibile eseguirne un altro permettendo di massimizzare in questo modo il carico di lavoro del processore.

1.3.2 Vector parallelism

Nel vector parallelism, il programma è composto da un singolo flusso di istruzioni che lavora su più dati contemporaneamente. Per far ciò, vengono adoperati:

- registri del processore denominati *vector register*, i quali possono contenere un insieme di dati;
- istruzioni specializzate per lavorare su dati vettoriali chiamate *vector instruction*.

Nei processori Intel l'insieme di istruzioni che lavorano su vettori sono chiamate AVX, mentre in quelli RISC-V esiste una specifica estensione dell'ISA che permette di lavorare su dati vettoriali.

1.4 Tassonomia delle architetture parallele

È possibile classificare le architetture parallele in più modi. Il più comune di questi è la tassonomia proposta da Flynn [11] nel 1966 (ed espansa nel 1972) nella quale vengono classificati dei sistemi in base alla tipologia di operazioni parallele che possono essere effettuate:

- SISD (*Single Instruction Single Data*): un processore, detto scalare, in grado di poter svolgere una singola istruzione su un singolo dato;
- SIMD (*Single Instruction Multiple Data*): un processore in grado di applicare un'operazione su un insieme di dati (vector parallelism);

- MIMD (*Multiple Instruction Multiple Data*): un processore in grado di esegure più operazioni distinte su gruppi di dati;
- MISD (*Multiple Instruction Multiple Data*): un processore in grado di esegure più operazioni su un singolo dato. Solitamente questa classificazione non risulta utilizzata.

Il concetto di MIMD può essere ulteriormente approfondito: l'esecuzione di più operazioni su dati distinti può essere caratterizzata sia dalla presenza di un processore multicore, che dall'utilizzo di un cluster di computer. Questa distinzione ci permette di definire ulteriori classi di architetture parallele, ovvero:

- architetture a memoria condivisa (*shared memory system*), in cui più unità di elaborazione lavorano contemporaneamente su uno stesso spazio di memoria;
- architetture a memoria distribuita (*distributed memory system*), in cui più unità di elaborazione sono collegate tra loro attraverso un network di interconnessione.

Un'ulteriore metodologia di classificazione delle architetture parallele, è legata alla modalità con la quale queste effettuano il context switching, ovvero la capacità di un processore di interrompere l'esecuzione di un lavoro (task) per poterne eseguire un altro. Il context switching è classificato in base al costo:

- *fine-grained context switching*, quando il costo per interrompere un'operazione è relativamente basso. La CPU in questo caso può interrompere periodicamente e per un breve periodo di tempo ciascun task;
- *coarse-grained context switching*, al contrario del primo, richiede un costo di interruzione dell'operazione non irrisorio. Viene adottato dai processori quando, per esempio, un task si trova in una fase di stallo (come il reperimento di dati in memoria) ed è conveniente per tale motivazione eseguire, nel mentre, altre operazioni.

1.5 Modelli di valutazione delle prestazioni

Un'architettura parallela può essere in grado di risolvere un'ampia varietà di problemi con caratteristiche molto differenti tra loro. Il criterio di valutazione secondo il quale le prestazioni di un sistema sono date dal solo tempo impiegato nell'eseguire degli applicativi, tende ad essere poco adatto in situazioni in cui vi è una suddivisione del lavoro irregolare o sono presenti delle forti dipendenze funzionali. È necessario adoperare dunque delle metriche astratte, indipendenti dalle caratteristiche del sistema e del problema considerato.

1.5.1 Speedup ed Efficienza

Nel parallelizzare un problema, vogliamo analizzare l'incremento avuto nell'impiegare più unità di calcolo (*worker*). Sia T_1 il tempo impiegato dal sistema nel risolvere un problema con un unico worker e $T(p)$ il tempo impiegato per risolverlo con p worker, si definisce *speedup*$(S(p))$ il rapporto:

$$S(p) = \frac{T_1}{T(p)} \tag{1.1}$$

In condizioni ottimali, nelle quali il problema è perfettamente scomponibile per il numero di worker e non vi è un overhead nel sincronizzarli, si ha $S(p) = p$, il quale è definito speedup lineare. Normalmente in situazioni realistiche si ha $S(p) \leq p$.

Ci aspettiamo, quindi, che il valore di S diminuisca all'aumentare del numero di unità di calcolo p. È possibile descrivere tale relazione dividendo S per p, ottenendo in questo caso l'efficienza E del programma:

$$E = \frac{S(p)}{p} = \frac{T_1}{T(p) \cdot p} \tag{1.2}$$

1.5.2 Legge di Amdahl

La legge di Amdahl è una metrica che permette di misurare le prestazioni di un sistema considerando la parte seriale del programma che viene eseguito.

Detto W_s il tempo di esecuzione della parte seriale del programma e W_p il tempo di esecuzione della restante parte parallelizzabile, possiamo ridefinire T_1 e $T(p)$ nel seguente modo:

- $T_1 = W_1 + W_p$
- $T(p) \geq W_1 + \frac{W_p}{p}$

Dunque, la formula dello speedup si modifica come segue:

$$S(p) \leq \frac{W_1 + W_p}{W_1 + \frac{W_p}{p}} \tag{1.3}$$

In particolare, sia f la percentuale seriale del programma, possiamo riscrivere T_p come:

$$T(p) \geq f \cdot T_1 + \frac{(1-f)T_1}{p}$$

Questo particolare ci permette di osservare che lo speedup è condizionato dalla percentuale non parallelizzabile del programma in quanto :

$$\begin{aligned} S(p) &= \frac{T_1}{T(p)} \\ &= \frac{T_1}{f \cdot T_1 + \frac{(1-f)T_1}{p}} \\ &= \frac{1}{f + \frac{1-f}{p}} \end{aligned} \tag{1.4}$$

Ponendo $p \to \infty$, 1.5 diventa:

$$S(p) = \frac{1}{f}$$

1.5.3 Legge di Gustafson-Barsis

Mentre nella Legge di Amdahl lo speedup è ottenuto incrementando il solo quantitativo di unità di calcolo mantenendo fissa la dimensione del problema, la legge di Gustafson-Barsis determina lo speedup all'incrementare

sia del numero p di processori sia della dimensione del problema. Data f la percentuale seriale del programma e p il numero di processori, lo speedup S è calcolato nel modo seguente:

$$\begin{aligned} S(p) &= f + p \cdot (1 - f) \\ &= p - f \cdot (p - 1) \end{aligned} \tag{1.5}$$

Capitolo 2

Processori RISC in ambito HPC

Con il termine *Reduced Instruction Set Computer* (RISC) si indica un principio di progettazione di microprocessori aventi come caratteristica principale un'architettura semplice il cui set di istruzioni (*Instruction Set Architecture*, ISA) è composto da istruzioni elementari, di dimensione uniforme, che vengono solitamente eseguite in un singolo ciclo di clock.

Questa filosofia si contrappone con quella CISC (*Complex Instruction Set Computer*), la quale prevede un'architettura hardware molto più complessa ed un ISA composto da istruzioni che tendenzialmente vengono eseguite in più cicli di clock.

Negli ultimi anni in ambito HPC sono stati principalmente utilizzati processori CISC, tuttavia la necessità di dover considerare anche l'aspetto energetico ha contribuito alla nascita di cluster basati su processori RISC, ad esempio con architettura RISC-V (*RISC FIVE*) e ARM (*Advanced RISC Machine*).

2.1 Impiego di CISC e RISC per l'HPC

Il profondo divario esistente tra le architetture CISC e RISC è stato argomento di molteplici dibattiti riguardo il loro impiego in contesti relativi all'High Performance Computing.

Agli inizi del 1990 il design dei processori era fortemente incentrato sull'aspetto prestazionale: l'architettura delle CPU e l'ISA creato per queste era focalizzato sulla necessità di renderle in grado di svolgere il maggior numero di istruzioni nel minor tempo possibile.

Dal 1990 al 2000 i sistemi HPC erano prevalentemente costituiti da architetture vettoriali e processori RISC adoperati nelle workstation del tempo, ovvero DEC Alpha, SPARC e MIPS. Dagli inizi del 2000 in poi queste vennero gradualmente sostituite da processori con architettura x86, i quali prevalsero all'interno della classifica TOP500 [36], nella quale vengono elencati i 500 sistemi cluster più potenti al mondo disponibili a livello commerciale [32].

L'avvento di tecnologie portatili, come laptop, cellulari e tablet, permise solo in un secondo momento una maggiore evoluzione dei processori RISC ed una conseguente reintroduzione di questi in contesti HPC, nei quali vi era un'esigenza crescente nel dover creare sistemi ad elevate prestazioni adoperando una politica di consumo energetico efficiente.

2.2 Advanced Risc Machine

L' Advanced Risc Machine (ARM) è una famiglia di processori RISC progettata dalla multinazionale britannica Arm holding, la cui attività principale è la vendita di licenze di produzione dei processori stessi.

Le cpu ARM sono basate su un design mirato ad un basso consumo energetico e permettono di offrire un ottimo compromesso tra prestazioni e costo. Date le loro caratteristiche, queste vennero originariamente adoperate per sistemi embedded a basso consumo; tuttavia, la loro evoluzione nel corso degli anni ha permesso di inserirle anche in sistemi richiedenti un grande quantitativo di potere di calcolo.

Esistono diversi sistemi HPC che hanno adottato tecnologia ARM:

- Mount-Blanc [29], il cui scopo è l'implementazione di cluster per HPC e Big data basati su processori ARM;
- il cluster di Huawei basato sulla cpu ARM Kunpeng 920 [21];
- il supercomputer Post-K [27].

2.3 RISC-V

RISC-V nasce nell'università di Berkley come progetto del Prof. Krste Asanović e degli studenti Yunsup Lee ed Andrew Waterman, che nel 2010 ne iniziarono a scrivere il set di istruzioni [33]. La necessità di ideare una nuova tipologia di processori derivava dall'esigenza di trovare delle ISA da adoperare nei futuri progetti dell'università.

Le scelte ai tempi erano ricadute principalmente nelle cpu x86 e ARM, le quali possedevano tuttavia un'architettura troppo complessa per gli scopi previsti ed erano protette da proprietà intellettuale. Le caratteristiche principali di RISC-V sono: la semplicità di implementazione, un design pulito e un ISA compatto e modulare.

2.3.1 Set di istruzioni

Uno dei punti di forza che contraddistingue RISC-V da altre famiglie di processori è l'ISA adoperato, che è in grado di svolgere operazioni suddivise in moduli distinti [23]. È possibile specificare una base a 32, 64 o 128 bit in grado di fornire funzioni che effettuano operazioni su interi che comprendono rotazioni, somme, sottrazioni, confronti e operatori booleani. I moduli che è possibile abilitare sono:

- **M**: moltiplicazione e divisione di interi;
- **A**: operazioni atomiche;

- **F, D, Q**: operazioni floating-point, le quali variano in base alla precisione che si vuole utilizzare;

- **C**: operazioni codificate di dimensione ridotta (da 32 bit a 16 bit).

L'astrazione e l'estensibilità fornita dalla modularità di RISC-V permette di specializzare un microprocessore in base alle richieste effettive della macchina, ma comporta lo svantaggio della non portabilità degli applicativi: un binario compilato in un sistema RISC-V con processori configurati per utilizzare determinati moduli, non potrebbe essere eseguito su altre macchine RISC-V configurate diversamente.

2.3.2 Design

La progettazione dell'architettura delle specifiche tecniche di RISC-V è basata sull'evoluzione nel tempo delle CPU RISC. Nella realizzazione sono state analizzate quelle che sono state delle scelte architetturali errate e si è cercato di dare rilievo ai punti di forza adottati dai processori RISC. RISC-V è un'architettura che, al contrario di ARM, è esente da proprietà intellettuale e le sue specifiche sono aperte alla comunità. Questo aspetto favorisce un continuo riscontro da parte di un bacino di utenti molto esteso ed agevola l'utilizzo dei processori nella costruzione di sistemi informatici.

2.4 Software stack

Oltre ad una componente hardware, un sistema informatico di qualunque tipo necessita di un insieme di strumenti che permettono la compilazione e l'esecuzione di programmi su di esso. In ambito HPC è necessario definire: un sistema operativo, un compilatore, librerie per il calcolo scientifico e librerie per la programmazione distribuita.

2.4.1 Compilatori

Un compilatore è un software in grado di ricevere in input una serie di istruzioni scritte in un determinato linguaggio di programmazione e tradurle in un secondo linguaggio o in una forma tale che possano essere eseguite direttamente dal processore della macchina.

Diversi compilatori forniscono un supporto per Arm. Per la compilazione di programmi scritti in C i più comuni risultano essere GCC [14](*GNU Compiler Collection*) ed *Arm Compiler*.

RISC-V risulta essere supportato dai compilatori GCC e LLVM; come mostrato in [22], questi sono in grado di generare file binari aventi la stessa dimensione eseguibili con circa gli stessi cicli di clock.

2.4.2 Librerie per calcolo scientifico

Una libreria per calcolo scientifico è una raccolta di funzioni ottimizzate per calcoli numerici. Questa fornisce un'interfaccia (API) ai programmi che necessitano di eseguire operazioni algebriche mettendo a disposizione funzioni che le implementano in maniera efficiente. La maggior parte delle librerie scientifiche sono basate sulla specifica BLAS [4](*Basic Linear Algebra Subprograms*), la quale definisce tre livelli di funzioni:

- livello 1, che comprende operazioni su scalari e vettori;
- livello 2, che comprende operazioni i cui operandi sono una matrice ed un vettore;
- livello 3, che comprende operazioni i cui operandi sono due matrici.

Sia Arm che RISC-V supportano diverse librerie open source come: *OpenBlas* [26], *Eigen* [10], *Blasfeo* [5] e *Blis* [6].

2.4.3 Librerie per programmazione parallela

Le librerie per la programmazione parallela sono una componente fondamentale di un sistema HPC in quanto forniscono ai programmi un'interfaccia che permette di parallelizzare l'esecuzione di istruzioni.

Tipicamente, in base al paradigma di programmazione che si vuole applicare, è possibile scegliere tra diverse alternative come *Open Multiprocessing* (OMP) e *Open Message Passing Interface* (OMPI) entrambe supportate dalle architetture Arm e RISC-V.

Open Multiprocessing

Open Multiprocessing (OMP) è una libreria che fornisce un supporto per lo sviluppo di applicazioni su sistemi a memoria condivisa. Il funzionamento è basato su un modello *fork-join*, nel quale il programma viene suddiviso in parti seriali e in parti da eseguire parallelamente. Le parti seriali del programma saranno eseguite da un unico processo chiamato *master thread*. Nel momento in cui sono raggiunte porzioni di codice dichiarate parallele, verrà creato un insieme di thread (*thread pool*) ciascuno dei quali eseguirà le istruzioni presenti nel blocco. Al termine dell'esecuzione della sezione parallela i thread precedentemente creati si sincronizzeranno tra loro ed il programma proseguirà con l'esecuzione del solo master thread.

Open Message Passing Interface

Open Message Passing Interface (OMPI) è una libreria adoperata per lo sviluppo di applicazioni su sistemi a memoria distribuita. Viene utilizzata in contesti nei quali occorre sincronizzare più computer all'interno di una rete per distribuire il carico di lavoro di un programma.

Capitolo 3

Ambiente di lavoro

In questo capitolo viene presentato l'ambiente di lavoro in cui sono stati eseguiti i test: Monte Cimone [3]. Il progetto è nato con lo scopo di definire un sistema hardware e software basato sul processore RISC-V per applicazioni legate al calcolo ad alte prestazioni.

3.1 Hardware

Il cluster è composto da 4 unità computazionali (computing blades) E4 RV007, di dimensione $42.5cm(w) \times 4.44cm(h) \times 40cm(d)$, al cui interno sono collocate due schede Mini-ITX SiFive Freedom U740 SoC (*System on a chip*) ciascuna delle quali è collegata ad un alimentatore da 250W in modo tale da controllarne l'accensione e lo spegnimento in maniera indipendente. Una scheda SiFive Freedom U740 è composta da:

- quattro core eterogenei U74 RV64GCB a 64bit;
- 16GB di memoria DDR4;
- slot di espansione PCIe.

I core U74 RV64GCB sono eterogenei e superscalari; possiedono una pipeline *dual-issue in-order execution* la quale riesce ad eseguire un massimo

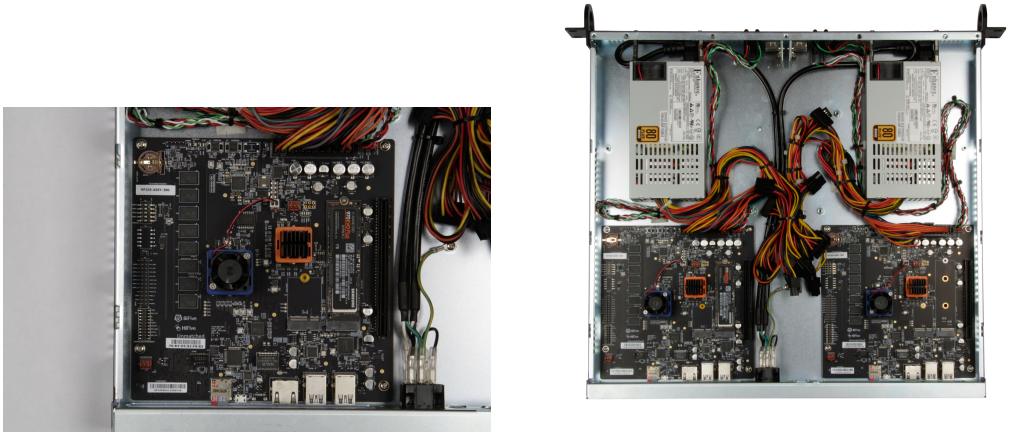


Figura 3.1: Configurazione hardware di un nodo del cluster di Monte Cimone.

di due istruzioni per ciclo di clock. In ciascun nodo è fornito un supporto per connessione Gigabit Ethernet dato dal chip Microsemi VSC8541. Per migliorare il throughput e la latenza, sono stati installati in due nodi degli adattatori Infiniband Mellanox ConnectX-4 FDR HCA.

3.2 Software

All'interno di Monte Cimone è stato installato uno stack software pensato per un ambiente HPC di produzione.

3.2.1 SPACK

Un sistema HPC di qualunque tipologia è composto solitamente da un vasto insieme di strumenti software, ciascuno dei quali necessita per un corretto funzionamento di configurazioni e dipendenze specifiche. Un approccio che prevede una gestione manuale di questi aspetti risulterebbe, oltre che molto complicato da gestire, anche poco scalabile. La soluzione adottata all'interno di Monte Cimone è stata quella di utilizzare Spack [12], un gestore

di pacchetti specializzato per sistemi HPC che permette di automatizzare il processo di installazione di software scientifico.

Il suo funzionamento è basato sulla creazione di un albero delle dipendenze, rappresentato da un grafo diretto e aciclico (DAG), il quale può essere generato da file di configurazione oppure tramite apposite istruzioni. Una dipendenza è definita all'interno di Spack tramite la creazione di classi Python che estendono la superclasse `Package`.

Spack supporta anche dipendenze virtuali: queste sono delle astrazioni di interfacce (come BLAS o MPI) che vengono fornite ai programmi al posto di specifiche implementazioni in modo tale da non creare dei vincoli di dipendenza stretti.

archspec

Molti gestori di pacchetti usano un approccio conservativo compilando programmi con istruzioni macchina generiche per favorire la portabilità a scapito dell'efficienza. Per tale ragione è stato integrato in Spack il modulo archspec [7], in grado di riconoscere e catalogare l'architettura sottostante e permettere, in fase di compilazione, di generare programmi eseguibili ottimizzati.

3.2.2 SLURM

SLURM è un software che permette la gestione del carico del lavoro in un cluster Linux:

- permette una preallocazione di risorse di calcolo per la durata di ciascun processo (job) avviato da un utente specifico;
- permette l'avvio, l'interruzione e il monitoraggio dei processi lanciati nel cluster;
- permette di gestire la coda di processi da lanciare all'interno del cluster.

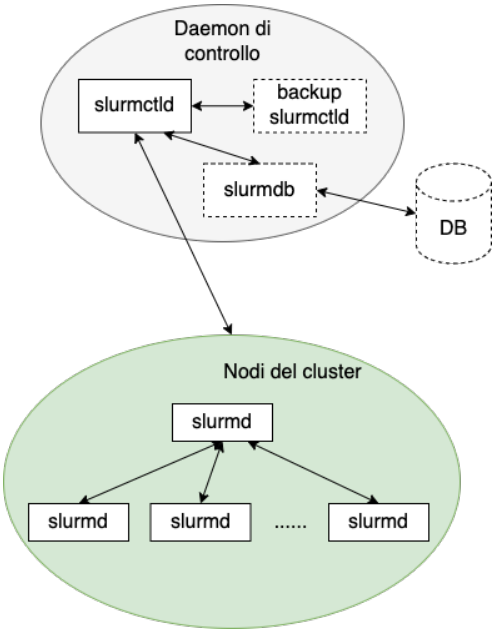


Figura 3.2: Schema funzionamento SLURM.

Il suo funzionamento è basato su un servizio centralizzato denominato slurmctld, eseguito su un nodo master (management node) il quale monitora le risorse disponibili e gestisce i jobs. In ogni altro nodo è eseguito un servizio slurmd che attende un input dal master per svolgere del lavoro.

Compilazione ed esecuzione di applicativi sul cluster

In fase di installazione è possibile configurare SLURM per riconoscere la presenza di MPI sulla macchina. Gli applicativi che adoperano la libreria OpenMPI potranno essere eseguiti in questo modo direttamente da SLURM tramite il comando `srun`. Attualmente questa opzione non è stata abilitata in Monte Cimone. Per la compilazione e l'esecuzione di applicativi all'interno del sistema è necessario creare dei file con estensione .sh contenenti delle direttive di configurazione per il comando `sbatch`. Di seguito viene riportato un esempio:

Listing 3.1: Esempio di file sbatch per l'esecuzione di un applicativo con mpi

```
#!/usr/bin/env bash

#SBATCH --nodes=8
#SBATCH --tasks-per-node=4
#SBATCH --cpus-per-task=1
#SBATCH --export=NONE

source /usr/share/modules/init/bash
module use /opt/share/modules/linux-ubuntu21.04-u74mc
module load openmpi/4.1.1/gcc-10.3.0-5hd3

OMPIPREFIX="$(dirname $(which mpirun))/../"
OMPIHOSTFILE=/opt/share/ompi.hostfile

mpirun \
        --prefix "${OMPIPREFIX}" \
        --hostfile "${OMPIHOSTFILE}" \
        -n 32 \
        my_exe
```

È possibile notare all'inizio delle direttive `#SBATCH`; queste permettono di specificare delle configurazioni di lancio per SLURM; in particolare le prime tre indicano rispettivamente:

- Il numero di nodi richiesto per l'esecuzione dell'applicativo (`--nodes`);
- Il numero di task che devono essere eseguiti in ciascun nodo (`--tasks-per-node`);
- Il numero di CPU impiegate per ciascun task (`--cpus-per-task`).

Successivamente l'applicativo viene lanciato attraverso il comando `mpirun` nel quale viene specificato un file di configurazione (hostfile) ed il numero di processi (questo ottenuto in base alle configurazioni riportate sopra).

Per la compilazione si adopera una configurazione analoga: un singolo nodo viene indicato come risorsa di calcolo, mediante inclusione di librerie

necessarie al programma per essere compilato (come ad esempio OpenMPI e cmake) ed infine si lancia il processo di compilazione.

3.3 Configurazione cluster

Monte Cimone è composto da 10 nodi:

- 8 nodi fisici che eseguono le operazioni;
- 2 nodi virtualizzati che si occupano dell'autenticazione degli utenti (*Login Node*) e della raccolta d'informazioni (*Master Node*).

I nodi Login e Master sono virtualizzati tramite l'infrastruttura KVM [31] (*Kernel-based Virtual Machine*) all'interno di un server denominato *Beta*. Questo permette di esporre i nodi alla rete esterna fornendo servizi di *Firewall*, *NAT* e *DHCP*.

Per poter lavorare con il cluster è necessario connettersi tramite ssh al nodo di login il quale consente di autenticare, tramite credenziali, un utente nel sistema fornendogli successivamente un ambiente dove poter adoperare SLURM e SPACK.

Il nodo Master viene impiegato per monitorare lo stato del cluster tramite l'utilizzo dei framework *Nagios* [25] e *Ganglia* [13]. Questi programmi permettono di usufruire di un'interfaccia web per la visualizzazione dello stato di ciascun nodo e per la raccolta di informazioni come il consumo energetico e calore generato.

Sono predisposti degli Hard Disk meccanici, configurati in RAID 5, i quali sono utilizzati dai nodi per il salvataggio dei dati tramite protocollo NFS (*Network File System*).

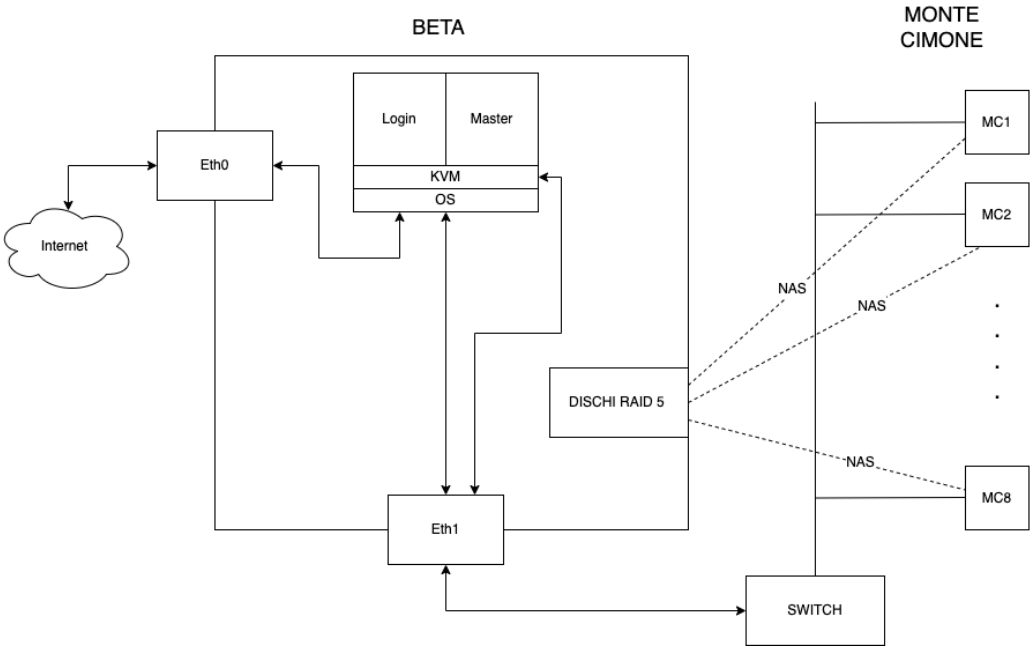


Figura 3.3: Schema di configurazione di Monte Cimone

Capitolo 4

Benchmark

Nei primi esperimenti condotti in [3] sono state valutate le prestazioni di Monte Cimone tramite l'utilizzo di tre benchmark: *HPL* [19], *QuantumEspresso* [30] e *STREAM* [35].

- HPL è un'implementazione di Linpack per architetture a memoria distribuita; il suo funzionamento prevede la risoluzione di sistemi lineari densi tramite la tecnica di fattorizzazione LU;

- QuantumEspresso è un insieme di software open source per la risoluzione di problemi di natura fisica come onde piane, funzionale della densità e pseudopotenziali;

- STREAM è un benchmark che calcola la larghezza di banda sostenibile all'interno di un nodo risolvendo problemi vettoriali.

I risultati ottenuti [3] hanno mostrato la capacità del sistema di raggiungere il 39.5% delle prestazioni massime di picco, lanciando un task per core fisico in tutti gli 8 nodi, ed il 36% delle prestazioni di picco ottenibile dall'unità per le operazioni sull'unità in virgola mobile (*Floating Point Unit*, FPU), con un consumo energetico compreso tra 4.810 e i 5.935 Watt nelle attività di calcolo più intense.

Vogliamo ora estendere lo studio legato alle prestazioni calcolando la capacità di Monte Cimone nel risolvere problemi che includono aspetti di comunicazione tra nodi e distribuzione non uniforme di dati nella memoria.

4.1 Graph500

Il benchmark *Graph500* [1] analizza la capacità di sistema HPC nel risolvere problemi su grafi. I passi che esegue sono:

- generazione del grafo (Kernel 1);
- visita in ampiezza (*Breadth First Search*, BFS) e calcolo dell'albero di visita (Kernel 2);
- cammini minimi da sorgente singola (*Single Source Shortes Path*, SSSP) per la ricerca del percorso più corto da un nodo verso tutti i nodi raggiungibili (Kernel 3).

Il funzionamento di Graph500 è presentato nell'algormito 1.
I dati da fornire in input sono:

- *S scale factor*, ovvero il fattore di scala rappresentante il logaritmo in base due del numero di vertici;
- *EF edege factor*, ovvero il rapporto tra numero di archi ed il numero di vertici.

Questi permettono di definire le classi del problema [17], ognuna delle quali rappresenta una specifica dimensione del grafo adoperato.

4.1.1 Kernel 1: Generazione del grafo

Il processo di generazione del grafo è suddiviso in due fasi: nella prima viene creata la lista degli archi, ciascuno rappresentato da una tupla $\langle V_s, V_e, W \rangle$

Algorithm 1: Graph500

Data: S, EF

Output: Informazioni riguardo le prestazioni misurate

```
1  Generazione della lista di archi;
2  Costruzione del grafo a partire dalla lista di archi (Kernel 1) ;
3  SK ← ∅;
4  for i = 0 to 63 do
5  |  Selezione di una chiave di ricerca k casuale tra i vertici del grafo;
6  |  SK ← SK ∪ {k};
7  end
8  foreach k ∈ SK do
9  |  Calcolo array predecessori T_k di k (Kernel 2);
10 |  Validazione di T_k
11 end
12 foreach k ∈ SK do
13 |  Calcolo array predecessori T_k e delle distanze D_k di k (Kernel
   |   3);
14 |  Validazione di T_k e D_k
15 end
```

in cui V_s e V_e indicano rispettivamente il vertice di partenza e di arrivo dell'arco, mentre W rappresenta il peso. Nella seconda fase viene adoperato un generatore di Kronecker [9] il quale crea quattro partizioni all'interno di una matrice di adiacenza aventi la stessa dimensione; successivamente distribuisce i vertici al suo interno con probabilità differenti. Siano A, B, C, D le quattro partizioni della matrice di adiacenza, le probabilità sono così definite: $A = 0.59\%$, $B = 0.19\%$, $C = 0.19\%$, $D = 0.05\%$. Una volta terminata la generazione del grafo in cui condurre le prove, vengono selezionati 64 vertici casuali che saranno utilizzati come chiavi di ricerca per BFS e SSSP.

4.1.2 Kernel 2: Ricerca in ampiezza

Il Kernel 2 calcola le prestazioni del sistema nell'effettuare una visita in ampiezza all'interno del grafo. L'algoritmo parte da un vertice sorgente s ed inizia a scoprire ricorsivamente tutti gli altri vertici k adiacenti costruendo un albero (*breadth-first tree*) avente s come radice. Il processo viene replicato per ciascuna delle chiavi di ricerca generate nel passaggio successivo alla generazione del grafo.

Algorithm 2: Visita in ampiezza

```
   Data: G, s ∈ G.V
 1 foreach v ∈ G.V − {s} do
 2 |   v.d ← ∞ ;
 3 |   v.π ← NIL ;
 4 end
 5 v.d ← 0;
 6 Q ← ∅;
 7 Enqueue(Q, s) ;
 8 while Q ≠ ∅ do
 9 |   u ← Dequeue(Q);
10 |   foreach v ∈ G.Adj(u) do
11 |   |   if v non è esplorato then
12 |   |   |   v.d = u.d + 1 ;
13 |   |   |   v.π ← u ;
14 |   |   |   Enqueue(Q, v) ;
15 |   |   end
16 |   end
17 end
```

4.1.3 Kernel 3: Cammini minimi da sorgente singola

Il Kernel 3 calcola le prestazioni del sistema nel determinare i cammini minimi in un grafo a partire da un nodo sorgente s. Data una funzione $w(p)$ che permette di definire il peso di un percorso p che congiunge due vertici u e v di un grafo G, si definisce $\delta(u, v)$ funzione del peso minimo tale per cui:

$$\delta(u,v) = \begin{cases} min\{w(p) : u \to v\} & \text{se esiste un percorso } p \text{ tra } u \text{ e } v \\ \infty & \text{altrimenti} \end{cases} \tag{4.1}$$

Un cammino minimo tra due vertici u, v è un percorso avente come peso $\delta(u, v)$. Graph500 implementa una versione parallela dell'algoritmo definita come Δ-*stepping* [24] (Delta stepping). Come per il Kernel 2 anche in questo caso i passi vengono ripetuti per ogni chiave di ricerca e verrà generata una struttura ad albero (*SSSP tree*) corrispondente ad ogni esecuzione.

4.1.4 Compilazione su Monte Cimone

L'implementazione del benchmark utilizzata per i test sul sistema è quella rilasciata da Graph500.org [18]. Viene fornito, insieme ai sorgenti degli applicativi, un *makefile* in grado di automatizzare il processo di compilazione. Per eseguirlo occorre creare un file avente estensione .sh contenente il comando `make` per l'avvio della compilazione, le direttive per l'inclusione dei moduli necessari (GCC e OpenMPI) resi disponibili da SPACK e le configurazioni per l'allocazione delle risorse nel cluster tramite SLURM.

Un primo tentativo di compilazione ha prodotto degli errori relativi a delle definizioni multiple di simboli aventi lo stesso nome. Questo era causato da un'incompatibilità dei sorgenti con la versione del compilatore GCC installata nel sistema (10.3.x) che utilizzava, come opzione predefinita, il flag `-fno-common` il quale impone al programma la generazione di una sola definizione per ciascun tentativo di creazione di variabile o oggetto [15].

Per risolvere il problema è stato necessario definire all'interno del makefile il flag `-fcommon`. Tramite questo ciascun simbolo viene creato in un blocco

comune e le definizioni uguali vengono unite in fase di collegamento dei file oggetto (link-time).

4.2 NAS Parallel Benchmarks

NAS Parallel Benchmarks (NPB) [2] è un insieme di benchmark che permette di valutare le prestazioni del sistema risolvendo problemi di fluidodinamica computazionale. NPB è composto da cinque applicazioni distinte (definite anche in questo caso "kernel"):

- IS (*Integer Sort*): vengono effettuate delle operazioni di ordinamento di interi testando sia la capacità del sistema di eseguire operazioni su questi sia le performance di comunicazione;
- EP (*Embarrassingly Parallel*): cerca di definire un limite superiore alle performance sulle operazioni su floating point;
- MG (*Multigrid*): permette di effettuare un'analisi delle comunicazioni a lunga e corta distanza;
- CG (*Conjugate Gradient*): viene calcolata l'approssimazione del più grande autovettore di una matrice sparsa, simmetrica e positiva adoperando la tecnica del gradiente coniugato;
- FT (*Fourier Transform*): calcola la trasformata 3D discreta di Fourier stimando le performance di comunicazione a lunga distanza.

Volendo focalizzarsi sull'aspetto legato alla comunicazione tra nodi e l'accesso in memoria, i kernel selezionati per essere eseguiti su Monte Cimone sono: IS, MG, CG, FT. Similmente a quanto visto per Graph500, anche in NPB sono definite delle classi [28] che permettono di definire le dimensioni del problema per ciascun kernel.

4.2.1 Integer Sort

Nel kernel IS vengono ordinati N numeri interi definiti tramite un generatore numerico pseudo casuale. I numeri creati sono distribuiti uniformemente nella memoria del sistema che esegue l'ordinamento adoperando una versione parallela del algoritmo *Bucket Sort* [34]. Nel caso di sistemi a memoria condivisa i numeri sono collocati in posizioni di memoria contigua mentre, per sistemi a memoria distribuita, ciascuna unità di lavoro terrà in memoria esattamente $\frac{N}{p}$ numeri, dove p è il totale di unità di lavoro presenti nel cluster.

4.2.2 Multigrid

Il kernel MG effettua delle iterazioni dell'algoritmo *V-cycle multigrid* [2] il quale permette di risolvere delle equazioni differenziali utilizzando delle discretizzazioni. Il valore ottenuto è un'approssimazione della soluzione u dell'equazione di Poisson:

$$\nabla^2 u = v \tag{4.2}$$

In una matrice di dimensione variabile in base alla classe del problema. Nell'equazione 4.2 il simbolo ∇^2 è definito come Laplaciano, un operatore differenziale che indica la divergenza del gradiente di una funzione in uno spazio euclideo.

4.2.3 Conjugate Gradient

Il kernel CG permette di calcolare una stima del più grande autovalore in una matrice sparsa, simmetrica e positiva tramite il metodo delle potenze inverse (Algoritmo 3).

$\|r\|$ indica la norma del vettore r il cui valore è calcolato nel seguente modo: $\|r\| = \sqrt{r^T r}$. La soluzione del sistema $Az = x$ è approssimata dal metodo del gradiente coniugato mentre i valori del numero di iterazioni L e lo scostamento λ sono definiti in base alla classe del problema.

Algorithm 3: Potenze inverse

Data: Matrice A di ordine n

1 $x \leftarrow [1, 1, \ldots, 1]^T$;
2 **for** $it = 1$ **to** L **do**
3 $\quad \|r\| \leftarrow ConjugateGradient(Az = x)$;
4 $\quad \zeta \leftarrow \lambda + 1/(x^t z)$;
5 $\quad$ Stampa it, ζ, $\|r\|$;
6 $\quad x \leftarrow z/\|z\|$;
7 **end**

4.2.4 Fourier Transform

Il kernel FT permette di effettuare il calcolo di una trasformata di Fourier per risolvere delle equazioni differenziali parziali. Viene discretizzato il valore della derivata parziale:

$$\frac{\partial v(z,t)}{\partial t} = -4\alpha\pi^2|z|^2 v(z,t) \tag{4.3}$$

In cui $v(z,t)$ ha valore :

$$v(z,t) = e^{-4\alpha\pi^2|z|^2 t} v(z,0) \tag{4.4}$$

applicando la trasformata discreta di Fourier 3-D e l'inversa.

4.2.5 Compilazione su Monte Cimone

Per la compilazione di NAS su Monte Cimone, i passaggi da effettuare sono analoghi a quanto visto per Graph500 nel paragrafo 4.1.4. Nella cartella `config`, presente al interno di quella contenente i sorgenti, occorre definire due file:

- `suite.def` che permette di specificare l'insieme di kernel che vogliono essere generati, assieme alle rispettive classi;

- `make.def` che permette di specificare delle configurazioni utilizzate in fase di compilazione.

Una volta creati i file di configurazione si può procedere nell'eseguire la compilazione, tramite l'utilizzo del comando `make`, utilizzando SLURM e SPACK come visto in precedenza.

Durante il procedimento è sorto un problema dato dalla versione di GCC in uso: venivano segnalate come errori alcune operazioni, eseguite dai kernel scritti in linguaggio Fortran, che utilizzavano degli operandi di tipo diverso tra loro. È stato necessario specificare all'interno di `config/make.def` il flag `-fallow-argument-mismatch` il quale permette di compilare i sorgenti trasformando gli errori apparsi in avvisi.

4.3 HPCBENCH

HPCBench [20] è un insieme di benchmark per sistemi HPC, scritto in liguaggio C, che permette di valutare le prestazioni della rete a cui i nodi sono connessi attraverso l'utilizzo di 3 tipologie di test: test per connessioni UDP (*User Datagram Protocol*), test per connessioni TCP(*Trasmission Control Protocol*) e test per comunicazioni MPI.

4.3.1 Descrizione dei test

Nei test per UDP e TCP due nodi del cluster agiscono da server e da client. Il client invia dei pacchetti al server mentre quest'ultimo rimane in ascolto per riceverli. Nella connessione sono utilizzati due canali di comunicazione :

- un canale instaura una connessione TCP sicura per comunicazioni di controllo;
- un canale permette il passaggio dei soli pacchetti di test.

Nel test per le comunicazioni MPI vengono analizzate le prestazioni nel passaggio di dati tra due nodi del cluster utilizzando delle direttive MPI.

4.3.2 Compilazione su Monte Cimone

Anche in questo caso, esattamente come visto in precedenza sia per Graph500 che per NAS Parallel Benchmark, i sorgenti di HPCBENCH presentano al loro interno un makefile per la compilazione degli eseguibili. L'esecuzione del processo di compilazione è stato svolto con gli stessi passaggi visti nei paragrafi 4.1.4 e 4.2.5.

Capitolo 5

Risultati Esperimenti

In questo capitolo sono mostrati i risultati degli esperimenti effettuati eseguendo i benchmark Graph500, Nas Parallel Benchmark e HPCBENCH su Monte Cimone.

5.1 Graph500

Negli esperimenti relativi a Graph500 è stata analizzata la capacità del sistema nel risolvere dei problemi su grafi utilizzando un numero variabile di nodi di calcolo (1, 2, 4). Gli esperimenti sono stati condotti utilizzando tre valori per SCALE: 18, 20 e 23. Utilizzare valori minori di 18 non avrebbe garantito la generazione di un carico di lavoro sufficiente grande per valutare in maniera adeguata le prestazioni mentre l'utilizzo di valori superiori a 23 producono un consumo eccessivo di memoria. L'obiettivo è quello di valutare la capacita di calcolo nel risolvere un problema di un singolo nodo e determinare la scalabilità del sistema all'aumentare delle risorse.

Per ogni test gli applicativi sono stati eseguiti nel seguente modo:

```
mpirun -x TMPFILE=graph_file \
           -x REUSEFILE=1 \
           --prefix "${OMPIPREFIX}" \
           --hostfile "${OMPIHOSTFILE}" \
```

Nodi	Costruzione	BFS (TEPS)			SSSP (TEPS)		
		MIN	MAX	Media	MIN	MAX	Media
1	18.83s	1.57×10^6	1.63×10^6	1.60×10^6	1.05×10^6	1.47×10^6	1.22×10^6
2	13.27s	1.79×10^6	1.99×10^6	1.93×10^6	1.09×10^6	1.56×10^6	1.28×10^6
4	13.71s	2.70×10^6	3.09×10^6	3.00×10^6	1.53×10^6	2.14×10^6	1.79×10^6

Tabella 5.1: Tabella esecuzione dei test BFS e SSSP con fattore di scala 18.

Nodi	Costruzione	BFS (TEPS)			SSSP (TEPS)		
		MIN	MAX	Media	MIN	MAX	Media
1	43.74s	1.57×10^6	1.60×10^6	1.59×10^6	963521	1.40×10^6	1.11×10^6
2	39.40s	1.83×10^6	1.94×10^6	1.90×10^6	1.04×10^6	1.54×10^6	1.24×10^6
4	37.39s	2.84×10^6	3.14×10^6	3.08×10^6	1.63×10^6	2.40×10^6	1.91×10^6

Tabella 5.2: Tabella esecuzione dei test BFS e SSSP con fattore di scala 20.

```
-n x \
graph500_reference_bfs_sssp Scale
```

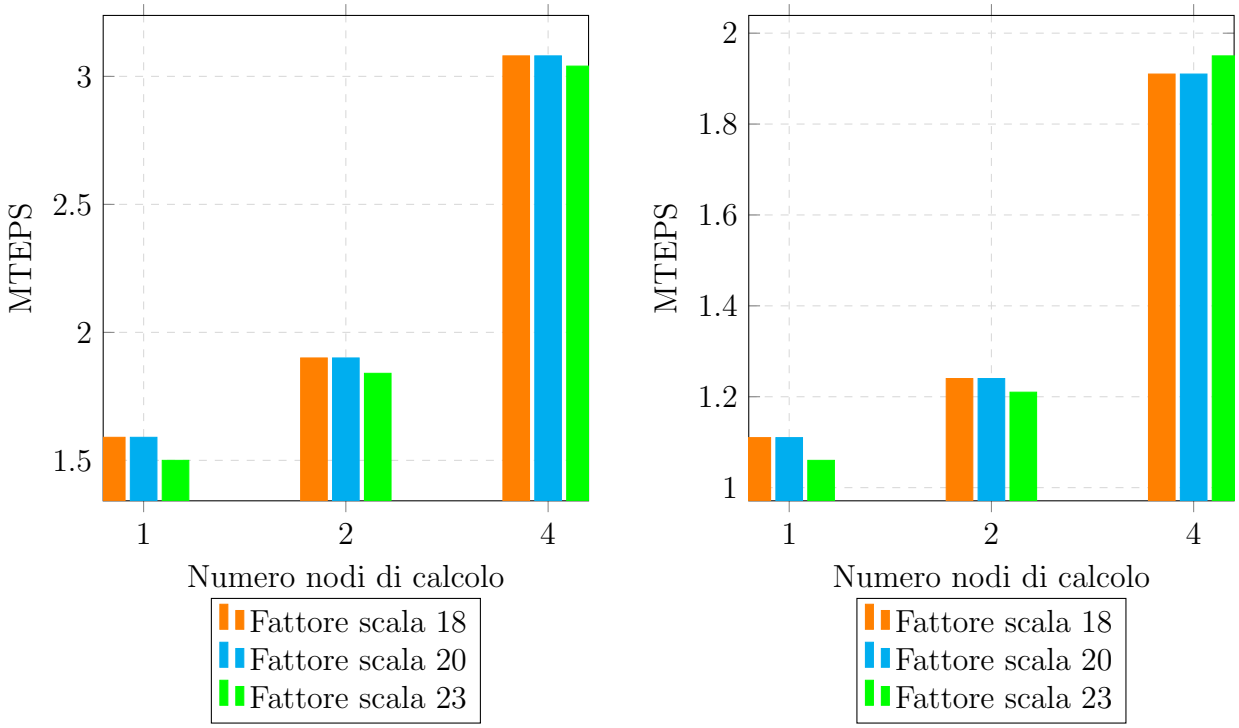
L'opzione `-x` permette di definire delle variabili d'ambiente per tutti i nodi: `TMPFILE` e `REUSEFILE`. Tramite queste viene specificato a Graph500 di salvare le informazioni del grafo in un file (`graph_file`) e di riutilizzarlo in fase di costruzione (kernel 1) nelle esecuzioni successive. L'unita di misura con cui sono riportati i risultati sono i TEPS (*Traversed Edges Per Second*) i quali rappresentano il numero di archi attraversati per secondo. Nell'esecuzione degli esperimenti con fattore di scala 23 è stato osservato che i test di validazione SSSP e BFS fallivano bloccando, di conseguenza, l'esecuzione del benchmark. Per risolvere il problema è stato necessario definire la variabile d'ambiente `SKIP_VALIDATION` con valore 1 tramite la quale viene impedito a Graph500 di effettuare la validazione degli alberi SSSP e BFS.

Il problema risulta scalare bene nel sistema all'aumentare del numero di risorse di calcolo disponibili. Questo è possibile vederlo sia dal tempo di

Nodi	Costruzione	BFS (TEPS)			SSSP (TEPS)		
		MIN	MAX	Media	MIN	MAX	Media
1	321.07s	1.44×10^6	1.51×10^6	1.50×10^6	885855	1.33×10^6	1.06×10^6
2	233.96s	1.83×10^6	1.85×10^6	1.84×10^6	1.01×10^6	1.51×10^6	1.21×10^6
4	148.41s	3.00×10^6	3.07×10^6	3.04×10^6	1.63×10^6	2.43×10^6	1.95×10^6

Tabella 5.3: Tabella esecuzione dei test BFS e SSSP con fattore di scala 23.

costruzione tendente a diminuire (soprattutto nel passaggio da 1 a 2 nodi) sia nelle prestazioni di BFS e SSSP il cui valore medio di TEPS calcolato aumenta gradualmente.



(a) Risultato esecuzione BFS

(b) Risultato esecuzione SSSP

Figura 5.1: Prestazioni di Graph500 con i fattori di scala utilizzati

5.2 NAS Parallel Benchmark

I programmi presenti nella suite di NAS Parallel Benchmark permettono di valutare la capacità del sistema di risolvere problemi che richiedono una comunicazione a lunga e corta distanza tra nodi. Lo scopo degli esperimenti effettuati è quello di analizzare le prestazioni del sistema nell'esecuzione dei kernel di NPB utilizzando il numero massimo di risorse disponibili. A causa di problemi all'infrastrutture di Monte Cimone, non è stato possibile effettuare test usando tutti gli 8 nodi di calcolo. Ogni kernel richiede un numero di processi pari ad una potenza di due. Avendo a disposizione un massimo di 7 nodi si è reso necessario adoperarne solamente 4 utilizzando dunque un numero di processi pari a 16. La classe utilizzata per ciascun problema è la C.

5.2.1 Integer Sort

Il tempo di esecuzione del test Integer Sort è 58.07 secondi. Sono state effettuate 10 iterazioni in cui, in ciascuna di queste, il numero di chiavi da ordinare è 134217728. Ciascun processo ha prodotto un risultato di 1.44 Mop/s ("op" indica l'operazione effettuata ovvero il key ranking), per un totale di 23.11 Mop/s raggiunto dal sistema.

5.2.2 Multigrid

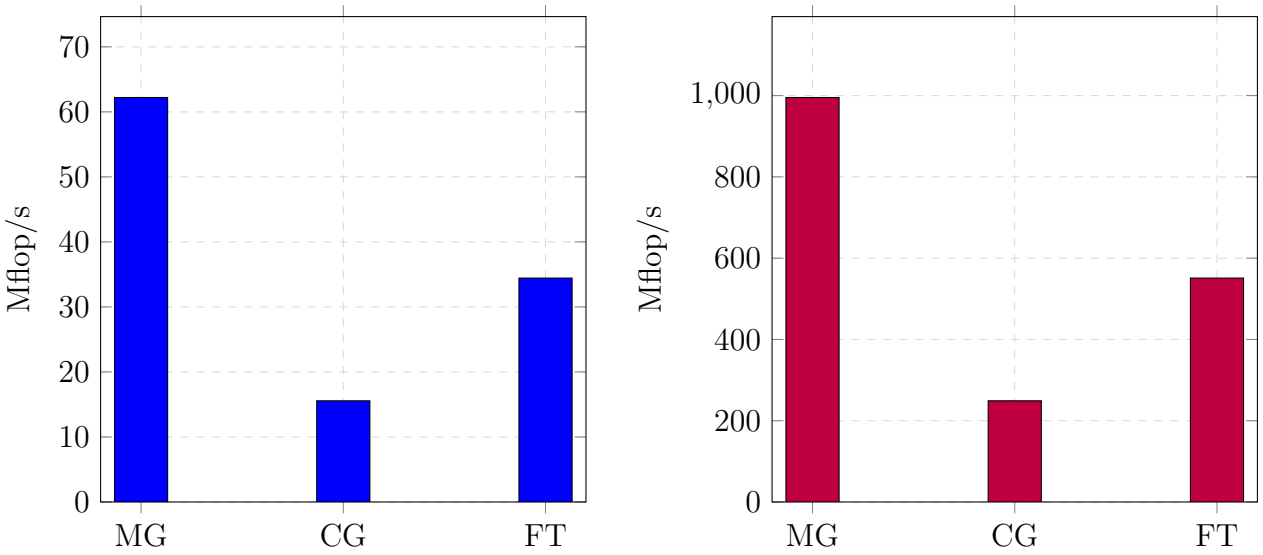
Il tempo di esecuzione registrato per il kernel Multigrid è di 156.42 secondi. Il test è stato effettuato adoperando le impostazioni di default ed utilizzando una matrice di dimensione $512 \times 512 \times 512$. Ciascun processo ha prodotto un risultato di 62.21 Mflop/s, per un totale di 995.33 Mflop/s raggiunto dal sistema.

5.2.3 Conjugate Gradient

Per Conjugate Gradient il tempo registrato è di 575.64 secondi. L'esecuzione del test è stata effettuata adoperando le impostazioni di default del kernel in cui la dimensione del sistema è uguale a 150000, numero di iterazioni uguale a 75 e valore di $\lambda = 110.000$. Ciascun processo ha prodotto un risultato di 15.56 Mflop/s, per un totale di 249.02 Mflop/s raggiunto dal sistema.

5.2.4 Fourier Transform

Il test effettuato con il kernel "Fourier Transform" è stato eseguito in un totale di 719.36 secondi. La dimensione della matrice adoperata è di $512 \times 512 \times 512$. Ciascun processo ha prodotto un risultato di 34.44 Mflop/s, per un totale di 551.03 Mflop/s raggiunto dal sistema.



(a) Prestazioni singolo processo

(b) Prestazioni complessive

Figura 5.2: Prestazioni dei benchmark MG, CG, FT in Mflop/s

Nodo 1	**Nodo 2**	**Throughput** Mbps	**Tempo medio** *s*
mcimone-node-7	mcimone-node-4	629.47	39.34
mcimone-node-1	mcimone-node-4	626.12	39.02
mcimone-node-7	mcimonde-node-1	622.53	40.13
mcimone-node-7	mcimone-node-5	613.78	40.57

Tabella 5.4: Tabella risultati HPCBENCH

5.3 HPCBENCH

Nei test HPCBENCH sono coinvolti solamente due nodi che comunicano tra loro facendo uso di direttive MPI. È necessario, ai fini di un corretto funzionamento, verificare che vengano eseguiti esattamente due processi e che questi siano presi in carico da nodi appartenenti a macchine diverse. Per far ciò occorre:

- Modificare l'hostfile indicando gli ip dei due nodi che si vogliono utilizzare, specificando che ognuno di questi possa ospitare un unico processo;
- Specificare nel file .sh utilizzato per l'esecuzione con `sbatch` le opzioni `--nodes=2` e `--tasks-per-node=1`.

I test sono stati eseguiti effettuando 15 prove nella quale i nodi effettuano 5 invii di pacchetti aventi dimensioni 300mb. I risultati ottenuti sono stati stimati da un insieme di dati raccolti in 3 esecuzioni.

A causa di problemi nel sistema, è stato possibile usare solo i nodi 1, 4, 5 e 7. I risultati riportati nella tabella 5.4 mostrano come la velocità di connessione si mantenga su valori costanti (622.78 Mbp/s di media).

Conclusioni

In questa tesi sono state analizzate le prestazioni del cluster RISC-V Monte Cimone tramite l'utilizzo dei benchmark Graph500, Nas Parallel Benchmark e HPCBENCH. Dai risultati ottenuti è emerso che:

- Il sistema risulta scalabile 5.1;
- Il sistema presenta delle scarse prestazioni nel reperire dati in posizioni irregolari della memoria (si veda 5.1, 5.2);
- I nodi del cluster riescono ad effettuare tra loro delle comunicazioni a lunga e corta distanza in maniera efficiente (si veda 5.2, 5.3).

L'utilizzo del sistema ha permesso di valutare lo stato di maturità di Monte Cimone: il cluster è risultato instabile nell'eseguire applicativi al suo interno; in diverse occasioni sono state riscontrate delle anomalie software che hanno reso impossibile l'utilizzo di tutte le risorse di calcolo. Questo aspetto ha inciso sull'analisi dei dati, in quanto non è stato possibile osservare il comportamento del sistema a pieno regime.

In futuro potrebbe risultare d'interesse, una volta raggiunta una completa stabilità del sistema, completare gli esperimenti effettuati eseguendo i test con tutte le risorse disponibili. Data la possibilità di installare acceleratori grafici (GPU) all'interno dei nodi del cluster, è possibile estendere l'analisi delle prestazioni risolvendo ulteriori tipologie di problemi come l'addestramento di modelli per l'intelligenza artificiale.

Appendice A

Utilizzo di Monte Cimone

In questa sezione sono mostrati i procedimenti necessari per lavorare all'interno dell'ambiente di Monte Cimone.

A.1 Connessione e creazione delle chiavi SSH

La connessione al server di Monte Cimone avviene adoperando il protocollo SSH. Una volta ottenute le credenziali di login è possibile collegarsi all'indirizzo del nodo di login utilizzando un client ssh o, in caso si stia lavorando in ambiente Linux, utilizzando il comando:

```
ssh username@MONTE_CIMONE_IP -p 2223
```

Eseguito l'accesso ci si ritroverà all'interno della cartella `home` dell'utente; sarà necessario a questo punto configurare delle chiavi SSH per poter lavorare su più nodi contemporaneamente:

1. Eseguire il comando `ssh-keygen` e seguire le istruzioni a schermo. Al completamento della procedura verranno create all'interno della cartella `~/.ssh/` i files `id_rsa` e `id_rsa.pub`;

2. Copiare il contenuto di `id_rsa.pub` all'interno del file `authorized_keys` presente sempre all'interno della cartella `~/.ssh/`;

3. Creare un nuovo file con estensione .sh avente il seguente contenuto:

Listing A.1: Fingerprint file.

```
#!/bin/bash

for i in {103..110}
do
        ssh-keyscan -t ecdsa-sha2-nistp256 \
        -H 192.168.1.$i >> ~/.ssh/known_hosts
done
```

4. Eseguire il file creato in precedenza. Questo procederà ad immettere all'interno di `~/.ssh/known_hosts` i fingerprint di tutti i nodi di Monte Cimone.

A.2 Compilare ed eseguire applicativi

Ogni script necessita di direttive che permettano di indicare il quantitativo di risorse che occorre utilizzare e i moduli necessari per l'esecuzione del programma.

Nel paragrafo 3.2 sono stati introdotti i software SPACK(3.2.1) e SLURM(3.2.2) i quali si occupano rispettivamente di gestire i moduli installati nel sistema e il carico di lavoro del cluster.

Le istruzioni che iniziano con l'espressione `#SBATCH` permettono di definire delle impostazioni per SLURM. Queste sono applicate al momento del lancio dello script con il comando `sbatch`. Per l'allocazione delle risorse è possibile specificare:

- Il numero di nodi che occorre adoperare (`#SBATCH --nodes=N`);
- Il numero di task che si vogliono allocare per nodo, (`#SBATCH --tasks-per-node= t`);
- Il numero di CPU per task (`#SBATCH --cpus-per-task=1`).

Listing A.2: File shell per la compilazione.

```
#!/usr/bin/env bash

#SBATCH --export=NONE
#SBATCH --job-name=JOB_NAME
#SBATCH --nodes=1
#SBATCH --exclusive
#SBATCH --output=%x_%A_%a.out
#SBATCH --error=%x_%A_%a.err

module use /opt/share/modules/linux-ubuntu21.04-u74mc
module load openmpi/4.1.1/gcc-10.3.0-5hd3
module load gcc/10.3.0/gcc-10.3.0-gyqt

make clean
make all
```

Lo script A.2 presenta un esempio di compilazione tramite makefile. Si noti che in questo caso l'utilizzo di un solo nodo risulta sufficiente.

Tramite la direttiva `#SBATCH --job-name=JOB_NAME` è possibile assegnare un nome al processo che sta per essere eseguito e con `#SBATCH --output=%x_%A_%a.err` e `#SBATCH --error=%x_%A_%a.out` sono generati due file contenenti l'output di `stderr` e `stdout` dell'esecuzione.

Il comando `module` permette di specificare i moduli spack che devono essere adoperati nell'esecuzione dello script. Per la compilazione è bastato includere il modulo relativo al compilatore `gcc` e `OpenMPI`.

Una lista completa dei moduli presenti all'interno del cluster è disponibile lanciando il comando `module avail`.

Listing A.3: Lancio di un applicativo

```
#!/usr/bin/env bash
```

```
#SBATCH --job-name=execute-graph500-job
#SBATCH --nodes= N
#SBATCH --export=NONE
#SBATCH --tasks-per-node=t
#SBATCH --cpus-per-task=1
#SBATCH --exclusive
#SBATCH --output=%x_%A_%a.out
#SBATCH --error=%x_%A_%a.err


module use /opt/share/modules/linux-ubuntu21.04-u74mc
module load openmpi/4.1.1/gcc-10.3.0-5hd3

OMPIPREFIX="$(dirname $(which mpirun))/../"
OMPIHOSTFILE=/opt/share/ompi.hostfile


mpirun \
        -x MY_ENV_VARIABLE = 1 \
        --prefix "${OMPIPREFIX}" \
        --hostfile "${OMPIHOSTFILE}" \
        -n x \
        executable
```

Nello script A.3 viene mostrato un esempio di esecuzione di un'applicazione tramite il comando `mpirun`. Il valore di x in `-n x` è determinato dal numero di nodi moltiplicato per il numero di task per nodi: $N \times t$. `--prefix` permette di specificare il prefisso delle installazioni di OpenMPI negli altri nodi, mentre `--hostfile` indica la posizione del hostfile nel quale sono presenti gli indirizzi IP di tutti i nodi all'interno della rete.

L'opzione `-x MY_ENV_VARIABLE = 1` definisce, in fase di esecuzione, una variabile d'ambiente di nome `MY_ENV_VARIABLE` avente valore 1.

Bibliografia

[1] James Ang et al. *Introducing the Graph 500*. Rapp. tecn. 2010. URL: https://www.researchgate.net/publication/255240580.

[2] D Bailey et al. *THE NAS PARALLEL BENCHMARKS*. Rapp. tecn. 1994.

[3] Andrea Bartolini et al. "Monte Cimone: Paving the Road for the First Generation of RISC-V High-Performance Computers". In: (mag. 2022). URL: http://arxiv.org/abs/2205.03725.

[4] *Basic Linear Algebra Subprograms*. URL: https://netlib.org/blas/.

[5] *Blasfeo: BLAS For Embedded Optimization*. URL: https://github.com/giaf/blasfeo.

[6] *Blis framework*. URL: https://github.com/flame/blis.

[7] Massimiliano Culpo et al. *archspec: A library for detecting, labeling, and reasoning about microarchitectures*. Rapp. tecn.

[8] Sujay B. Desai et al. "MoS2 transistors with 1-nanometer gate lengths". In: *Science* 354.6308 (ott. 2016), pp. 99–102. ISSN: 0036-8075. DOI: 10.1126/science.aah4698.

[9] Jure@cs Stanford Edu et al. *Kronecker Graphs: An Approach to Modeling Networks Jure Leskovec Zoubin Ghahramani*. Rapp. tecn. 2010, pp. 985–1042.

[10] *Eigen: C++ template library for linear algebra*. URL: https://eigen.tuxfamily.org/index.php?title=Main_Page.

[11] Michael J. Flynn. *Very High-Speed Computing Systems*. 1966. DOI: 10.1109/PROC.1966.5273.

[12] Todd Gamblin et al. "The Spack package manager: Bringing order to HPC software chaos". In: *International Conference for High Performance Computing, Networking, Storage and Analysis, SC*. Vol. 15-20-November-2015. IEEE Computer Society, nov. 2015. ISBN: 9781450337236. DOI: 10.1145/2807591.2807623.

[13] "Ganglia". In: (). URL: http://ganglia.sourceforge.net/.

[14] *GCC, the GNU Compiler Collection*. URL: https://gcc.gnu.org/.

[15] *Gnu changes in version 10*. URL: https://gcc.gnu.org/gcc-10/changes.html#c.

[16] Gordon E. Moore. "The future of integrated electronics". In: ().

[17] *Graph500 Problem classes*. URL: https://graph500.org/?page_id=12#tbl:classes.

[18] *Graph500 repository*. URL: https://github.com/graph500/graph500.

[19] *HPL - A Portable Implementation of the High-Performance Linpack Benchmark for Distributed-Memory Computers*. URL: https://netlib.org/benchmark/hpl/.

[20] Ben Huang, Michael Bauer e Michael Katchabaw. *Network Performance in Distributed HPC Clusters*. Rapp. tecn.

[21] *Huawei Unveils "Industry's Highest-Performance ARM-based CPU"*. 2019. URL: https://insidehpc.com/2019/01/huawei-unveils-industrys-highest-performance-arm-based-cpu/.

[22] Institute of Electrical and Electronics Engineers. *2020 International Conference on Omni-Layer Intelligent Systems (COINS) : 31 August-2 September 2020, Barcelona, Spain*. ISBN: 9781728163710.

[23] David Kanter. *RISC-V OFFERS SIMPLE, MODULAR ISA New CPU Instruction Set Is Open and Extensible*. Rapp. tecn. 2016.

[24] U Meyer e P Sanders. *Stepping : A Parallel Single Source Shortest Path Algorithm*. Rapp. tecn.

[25] *Nagios*. URL: https://www.nagios.org/.

[26] *OpenBlas: An optimized BLAS library*. URL: https://www.openblas.net/.

[27] *Post-K: Building up Arm HPC Ecosystem*. Rapp. tecn. 2017.

[28] *Problem Sizes and Parameters in NAS Parallel Benchmarks*. URL: https://www.nas.nasa.gov/software/npb_problem_sizes.html.

[29] *Project Mount-Blanc*. 2011. URL: https://www.montblanc-project.eu/project.

[30] *Quantum Espresso*. URL: https://www.quantum-espresso.org/.

[31] Avi Kivity Qumranet et al. *kvm: the Linux Virtual Machine Monitor*. Rapp. tecn.

[32] Nikola Rajovic et al. "Supercomputing with commodity CPUs: Are mobile SoCs ready for HPC?" In: *International Conference for High Performance Computing, Networking, Storage and Analysis, SC*. IEEE Computer Society, 2013. ISBN: 9781450323789. DOI: 10.1145/2503210.2503281.

[33] *RISC-V History*. URL: https://riscv.org/about/history/.

[34] William Saphir Rob Van der Wijngaart Alex Woo Maurice Yarrow. *New Implementations and Results for the NAS Parallel Benchmarks 2*. Rapp. tecn. 2000. URL: http://www.nas.nasa.gov/NAS/NPB/..

[35] *STREAM: Sustainable Memory Bandwidth in High Performance Computers*. URL: https://www.cs.virginia.edu/stream/.

[36] *TOP500*. URL: https://www.top500.org/.

[37] Xingqi Zou et al. "Breaking the von Neumann bottleneck: architecture-level processing-in-memory technology". In: *Science China Information Sciences* 64.6 (giu. 2021), p. 160404. ISSN: 1674-733X. DOI: 10.1007/s11432-020-3227-1.