Metodi e strumenti per la valutazione delle prestazioni del software parallelo attraverso un caso di studio: il software ROMS

Simone Celestino

6 giugno 2018

Indice

Introduzione

L'obiettivo di questa tesi è quello individurare e studiare metodi e strumenti per la valutazione di software, a partire dalla valutazione di architetture che sfruttano il parallelismo [1], alle tecniche per il profiling [1] e valutazione di software eseguiti su architetture ad alte prestazioni (HPC); ciò viene fatto analizzando le prestazioni del codice sorgente evidenziando i colli di bottiglia che ne limitano l'efficienza e individuando i kernel computazionalmente più onerosi. Un aspetto necessario per la valutazione è il confronto tra la prestazione 'standard' della macchina (di picco e sostenuta), confrontandola con quella ottenuta dall'applicazione di interesse. Queste analisi possono suggerire come il software può essere eseguito in maniera efficiente e 'prestante' su le architetture analizzate [3].
Ci si pone le seguenti domande:

- Il sistema di calcolo a disposizione è adeguato all'esecuzione di questo software?
- Tale software raggiunge le prestazioni attese?

In Figura 1 si mettono in evidenza due macro-aree: una relativa al calcolo delle prestazioni di una architettura, l'altra allo studio delle prestazioni di un software. Occorre dare una risposta da un lato sui limiti del sistema di calcolo dall'altro sulle cause che portano un software a non utilizzare efficacemente l'architettura utilizzata, e quindi alle azioni da intraprendere per migliorarle. Da un lato si trovano gli strumenti per valutare un sistema di calcolo chiamati benchmark, dall'altro si trovano gli strumenti ed i modelli che rivolgono l'attenzione alla valutazione delle prestazioni ottenute dall'applicazione di interesse.
Ogni sistema di calcolo è costituito da elementi che vanno analizzati singolarmente. Sarà necessario porre l'attenzione su:

[1] Per profiling, si intende l'analisi dinamica volta a collezionare informazioni sul comportamento del software durante la sua esecuzione, ad es. lo spazio utilizzato, la complessità di tempo, frequenza d'uso di una particolare funzione etc. [2]

- Calcolo
- Comunicazione
- I/O
- Memoria

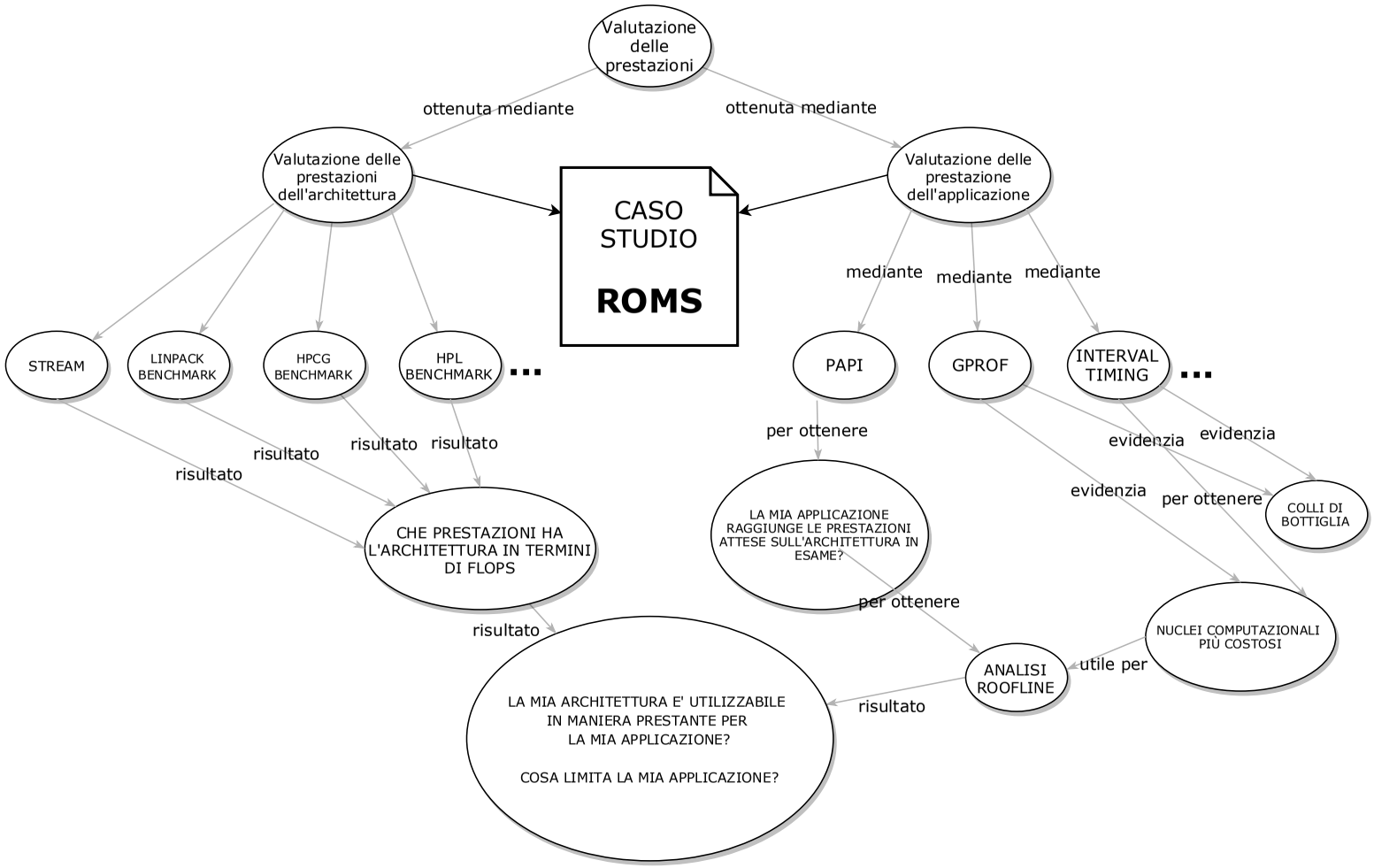


Figura 1: L'idea alla base di questo lavoro di tesi

Capitolo 1

Valutazione delle prestazioni

1.1 Concetti Premiminari e definizioni

In questo paragrafo introduttivo si analizzeranno rapidamente i concetti base utilizzati in questa tesi.
Il calcolatore preso in considerazione in questo studio è schematizzato a partire dal quello ideato da Von Neumann nel '45 [4] (Figura 1.1) che ha dato il via al concetto dell'attuale calcolatore [1] (Figura 1.2).

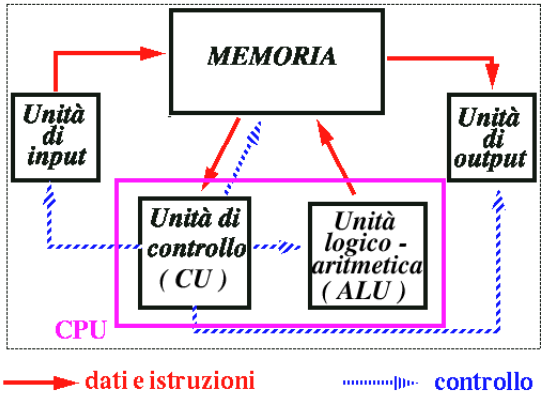


Figura 1.1: Il calcolatore schematizzato da Von Neumann nel '45

Si possono elencare in maniera più dettagliata le componenti (Figura 1.2):

- ALU: Esegue calcoli come addizioni e confronti
- FPU: Esegue operazioni su numeri a virgola mobile

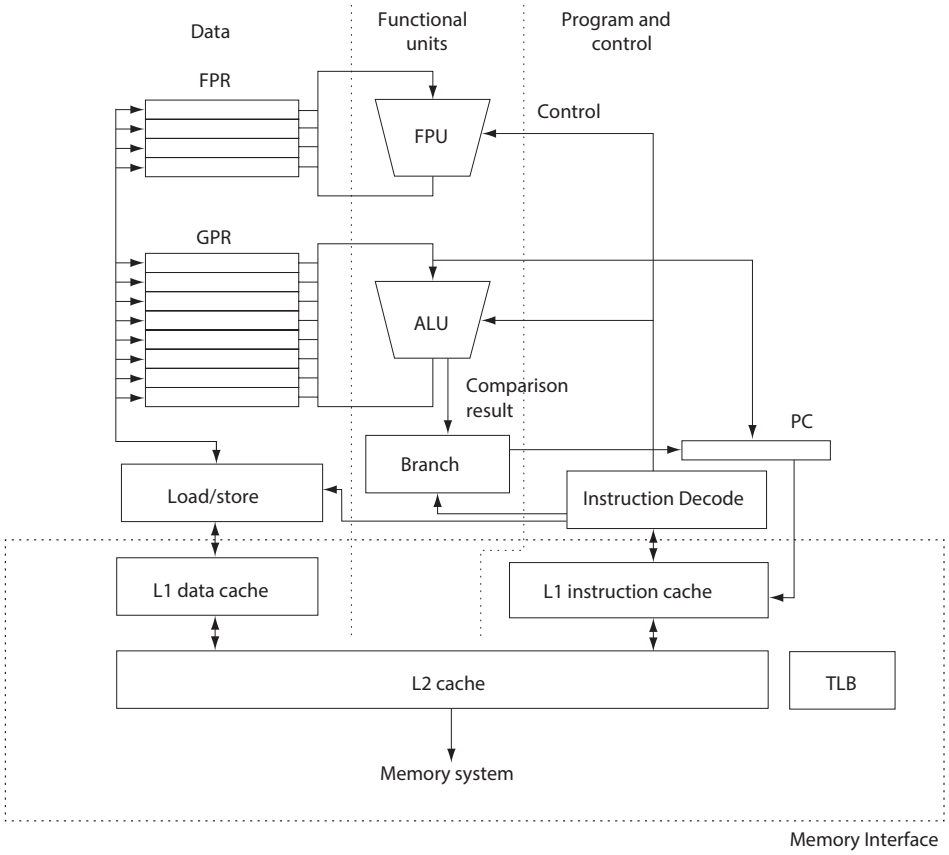


Figura 1.2: Una schematizzazione a grana più fine del calcolare schematizzato da Von Neumann

- Load/Store: Esegue operazioni di caricamento e memorizzazione sui dati
- Registri: Memorie veloci per memorizzare risultati intermedi sono suddivise a loro volta in registri floating-point (FPR) e registri per uso generico (GPR);
- Program Counter: Contiene l'indirizzo dell'istruzione in esecuzione (PC), è un registro interno alla CPU che conserva l'indirizzo di memoria della prossima istruzione da eseguire;
- Interfaccia di memoria: Fornisce l'accesso alla memoria del sistema;
- Memoria: Dove vengono immagazzinate le informazioni (Cache e Memoria principale);

Cosa si intende precisamente per prestazione? Osservando la tabella 1.1 quale di questi aeroplani ha le migliori prestazioni? Se per prestazione si

Aeroplano	*Capacità passeggeri*	*Velocità di crociera (Km/H)*	*Autonomia (Km)*
Boeing 777	375	980	7400
Boeing 747	470	980	6640
BAC/Sud Concorde	132	2160	6400
Douglas DC-8-50	146	870	13950

Tabella 1.1: Caratteristiche di ciascun aeroplano di cui ci interessa conoscere le prestazioni

intende la capacità di trasportare un singolo passeggero da un luogo ad un altro nel minor tempo possibile (quindi si considera il mezzo più veloce) allora la scelta migliore è il Concorde. Se invece si intende l'aereo con il maggior numero di passeggeri allora il migliore sarà il Boeing 747.
Volendo quindi stabilire la prestazione per un calcolatore, quali parametri vanno considerati? La prestazione di un calcolatore è la capacità di eseguire uno specifico compito in un determinato tempo. Ci si potrebbe riferire al:

- Tempo impiegato a fornire una risposta ad un compito assegnato (tempo di risposta);
- La quantità di lavoro eseguita nell'unità di tempo (throughput);
- Tempo in cui la CPU è impegnata nel calcolo.

si decide di valutare le prestazioni in termini di **tempo di risposta**. Gli aspetti su cui porre l'attenzione sono strettamente legate alle caratteristiche del processore, ovvero:

- Ciclo di clock
- Numero di operazioni floating-point per ciclo di clock

Definizione 1.1 (Clock) *Il clock è un dispositivo che produce impulsi ad intervalli regolari, può essere definito informalmente come l'orologio del calcolatore.*

Definizione 1.2 (Ciclo di clock) *Il ciclo (o periodo) di clock T_c è la misura del tempo che intercorre tra due colpi di clock successivi (Figura 1.3).*

Definizione 1.3 (Frequenza) *La frequenza f è il numero di colpi di clock che vengono eseguiti in una unità di tempo (Figura 1.4), si misura in Hertz (numero di cicli di clock eseguiti in un secondo) e può essere scritta anche come:*

$$f = \frac{1}{T_c}$$

Figura 1.3: Ciclo di clock

Definizione 1.4 (FLOPS) *Il FLOPS (FLOating Point per Second) è la unità utilizzata comunemente per le prestazioni di un calcolatore e definisce il numero di operazioni floating-point che vengono eseguite in un secondo [5].*

In questo contesto si parlerà di operazioni floating point in doppia precisione considerando una operazione in virgola mobile $(+ - \times /)$. [1]

[1] Talvolta viene utilizzata la misura del MIPS (Million Instruction Per Second, ovvero le milioni di istruzioni al secondo) ma la moltitudine di istruzioni esistenti e la diversa implementazione delle stesse tra le architetture l'ha resa una misura non affidabile.

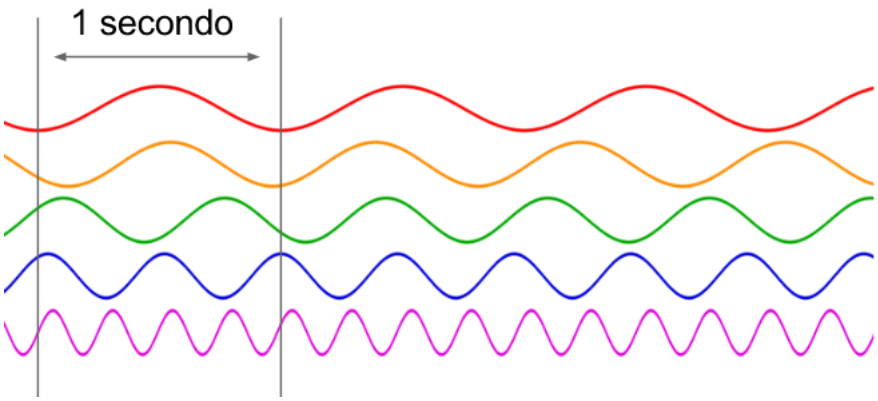


Figura 1.4: Frequenza

Definizione 1.5 (Numero di operazioni floating-point per ciclo di clock) *Si indica con N_c il numero di operazioni (floating-point) per ciclo di clock, definita in letteratura anche come clock-cycle concurrency [6].*

Definizione 1.6 (t_{flop}) *Il tempo per eseguire una operazione è indicato con t_{flop}.*

$$t_{flop} = \frac{T_c}{N_c}$$

Definizione 1.7 (Prestazione di picco) *La prestazione di picco P_{peak} indica il numero massimo di operazioni floating point (virgola mobile) che un sistema in una unità di tempo (il secondo) esegue. Risulta:*

$$P_{peak} = \frac{1}{t_{flop}}$$

ovvero [4]:

$$P_{peak} = \frac{N_c}{T_c} = f \times N_c \tag{1.1}$$

ed è misurata in FLOPS [7]. La prestazione di picco viene così definita nelle architetture multi-core:

$$P_{peak\_core} = P_{peak} \times C \tag{1.2}$$

dove C è il numero di core dell'architettura.
La complessità computazionale definisce il costo degli algoritmi in termini di tempo di elaborazione e la quantità di memoria utilizzata. In generale per misurare il costo computazionale di un algoritmo si definiscono

- Complessità di tempo $T(n)$;
- Complessità di spazio $S(n)$.

dove n è la dimensione del problema.

Definizione 1.8 (Complessità di tempo) *La complessità di tempo $T(n)$ di un algoritmo è la funzione che che esprime il numero di operazioni dominanti in dipendenza della dimensione computazionale n del problema [8].*

Esempio 1.9 *Si supponga di volere calcolare la complessità di tempo della moltiplicazione di due matrici di dimensione $n \times n$: $C = A \times B$. Per calcolare $C[i, j]$ si eseguono $2n$ letture, n moltiplicazioni, $n-1$ addizioni ed 1 scrittura. Per calcolare C occorrono quindi $n^2 \times$ ($2n$ letture, n moltiplicazioni, $n-1$ addizioni ed 1 scrittura), per cui:*

- $2n^3$ *letture*
- n^3 *moltiplicazioni*
- $n^2 * (n-1)$ *addizioni*
- n^2 *scritture*
- *Totale:* $T(n) = 2n^3 + n^3 + n^2 * (n-1) + n^2 = 4 * n^3$

Dato un algoritmo A la complessità viene determinata contando il numero di operazioni aritmetiche dell'algoritmo.

Definizione 1.10 (Complessità di spazio) *La complessità di spazio $S(n)$ di un algoritmo è la funzione che esprime la dimensione totale delle strutture dati utilizzate per memorizzare i dati di input, locali e di output, in dipendenza della dimensione computazionale n del problema [8].*

Esempio 1.11 *Si supponga di voler calcolare la complessità di spazio della moltiplicazione di due matrici $n \times n$: $C = A \times B$. Se si memorizzano tre matrici piene di dimensione $n \times n$ si ha:*

$$S(n) = n^2 + n^2 + n^2 = 3n^2$$

Misura dello spazio di memoria richiesto da un algoritmo

Per ottenere il quantitativo di memoria richiesto da un algoritmo è necessario innanzitutto valutare il massimo numero di elementi che siano memorizzabili dal sistema in uso, quindi:

$$N_e = \frac{R_{size}}{sizeof_{type}} \tag{1.3}$$

dove N_e è il quantitativo di dati che si ha necessità di memorizzare, R_{size} è il quantitativo di memoria disponibile sul sistema di calcolo in uso e $sizeof_{type}$ è la dimensione in byte che rappresenta il dato.

Esempio 1.12 *Ad esempio se avessimo un sistema di calcolo con 8GByte di RAM potremmo memorizzare un miliardo di numeri floating-point in doppia precisione (un numero floating-point in doppia precisione si rappresenta con 64 bit cioè 8 Byte):*

$$N_e = \frac{8.000.000.000}{8} = 1.000.000.000$$

Definizione 1.13 (Tempo dell'algoritmo) *Dato un algoritmo definiremo il tempo dell'algoritmo $T(A)$ come:*

$$T(A) = N_{flop} * t_{flop}$$

dove N_{flop} è il numero di operazioni dell'algoritmo.

Definizione 1.14 (Tempo del software) *Dato un algoritmo definiremo software la sua implementazione in uno specifico ambiente di calcolo.*
Detto $T_{sw}(A)$ il tempo di esecuzione del software risulterà sempre:

$$\frac{T(A)}{T_{sw}(A)} < 1$$

pertanto per eseguire l'algoritmo sarà necessario effettuare quantomeno, oltre alle operazioni aritimetiche, accessi per leggere e scrivere nella memoria che ovviamente richiedono tempo.
Infatti è possibile affermare che il tempo del software è almeno:

$$T_{sw}(A) = N_{mem} \times t_{mem} + N_{flop} \times t_{flop} \tag{1.4}$$

N_{mem} è il numero di accessi alla memoria e t_{mem} è il tempo per accedere ad un dato in memoria. Quindi:

Definizione 1.15 (Prestazione sostenuta) *Fissato il software $SW(A)$ che implementa l'algoritmo A, la prestazione sostenuta (o effettiva) P_{actual} è il numero di FLOPS che un calcolatore esegue nell'unità di tempo [7, p. 4] e si esprime come:*

$$P_{actual} = \frac{N_{flop}}{T_{sw}(A)} \tag{1.5}$$

Quindi:

$$P_{actual} = \frac{N_{flop}}{T_{sw}(A)} = \frac{N_{flop}}{N_{mem} \times t_{mem} + N_{flop} \times t_{flop}} = \frac{\frac{1}{t_{flop}}}{1 + (\frac{N_{mem}}{N_{flop}} \frac{t_{mem}}{t_{flop}})} = \frac{P_{peak}}{1 + (\frac{N_{mem}}{N_{flop}} \frac{t_{mem}}{t_{flop}})}$$

il fattore

$$\frac{N_{mem}}{N_{flop}} \frac{t_{mem}}{t_{flop}} \tag{1.6}$$

rappresenta il fattore di decadimento della prestazione dove:

Definizione 1.16 (Traffico parassita) *Si esprime come:*

$$\frac{N_{mem}}{N_{flop}} \tag{1.7}$$

e dipende dall'algoritmo. Mentre,

$$\frac{t_{mem}}{t_{flop}}$$

dipende dall'hardware.

Quindi ridurre il fattore (1.6) permette di avvicinare la prestazione sostenuta alla prestazione di picco.

Definizione 1.17 (Larghezza di banda di memoria) *Con il termine larghezza della banda di memoria (o bandwidth di memoria) si indica la quantità di dati che può transitare, mediante una connessione, in un dato periodo di tempo contemporanetamente.*

Definizione 1.18 (Prestazione di picco del bandwidth) *la prestazione di picco del bandwidth B_{peak} indica il numero massimo di accessi alla memoria, in termini di **Byte/s**, che un sistema può effettuare nell'unità di tempo. Risulta:*

$$B_{peak} = \frac{1}{t_{mem}} \tag{1.8}$$

Definizione 1.19 (Prestazione sostenuta del bandwitdh di memoria) *Fissato il software SW(A) che implementa l'algoritmo A, la prestazione sostenuta del bandiwidth è il numero di byte/s che un sistema effettua nell'unità di tempo, e si esprime come:*

$$B_{actual} = \frac{N_{mem}}{T_{SW}(A)} \tag{1.9}$$

Quindi

$$B_{actual} = \frac{N_{mem}}{T_{sw}(A)} = \frac{N_{mem}}{N_{mem} \times t_{mem} + N_{flop} \times t_{flop}} = \frac{\frac{1}{t_{mem}}}{1 + (\frac{N_{flop}}{N_{mem}} \frac{t_{flop}}{t_{mem}})} = \frac{B_{peak}}{1 + (\frac{N_{flop}}{N_{mem}} \frac{t_{flop}}{t_{mem}})}$$

il fattore

$$\frac{N_{flop}}{N_{mem}} \frac{t_{flop}}{t_{mem}} \tag{1.10}$$

rappresenta il fattore di decadimento della prestazione del bandwidth di memoria e:

$$\frac{N_{flop}}{N_{mem}}$$

è detto:

Definizione 1.20 (Intensità computazionale o aritmetica) *Indica il numero di operazioni effettuate per byte di traffico di memoria [9, p. 66] e si indica con:*

$$AI = \frac{N_{flop}}{N_{mem}} \tag{1.11}$$

tale fattore dipende dall'algoritmo mentre

$$\frac{t_{flop}}{t_{mem}}$$

dipende dall'hardware.

L'indice di intensità aritmetica è la stima del numero di richieste che un software effettua alla memoria, o meglio è il numero di operazioni in virgola mobile per byte di memoria.

Definizione 1.21 (Latenza) *Si definisce con latenza il tempo che intercorre tra la richiesta di un dato e la sua effettiva disponibilità.*

Definizione 1.22 (Cache miss) *Con cache miss (fallimento della cache) si intende la mancata lettura o scrittura di un dato nella memoria cache, ne consegue un aumento della latenza per reperire il dato nella memoria principale.*

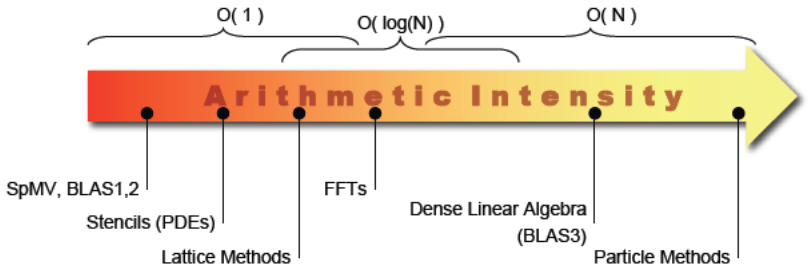


Figura 1.5: Intensità Aritmetica

Definizione 1.23 (Collo di bottiglia) *Il collo di bottiglia è un fenomeno che si verifica quando le prestazioni del sistema sono limitate da un singolo componente la cui presenza ne pregiudica l'efficienza.*

Definizione 1.24 (Tempo dell'algoritmo parallelo) *Se $T(A)$ è il tempo dell'algoritmo A, nel caso che tale algoritmo debba essere eseguito da una macchina parallela con p processori non tutte le operazioni possono essere suddivise ed eseguite concorrentemente tra i vari processori per cui si ha:*

$$T(A) = T_{seq}(A) + T_{par}(A) \tag{1.12}$$

con

$$T_{seq}(A) = N_{seq} \times t_{flop}$$

e

$$T_{par}(A) = N_{par} \times t_{flop}$$

dove N_{seq} è il numero di operazioni eseguite sequenzialmente e N_{par} è il numero di operazioni eseguite suddivise tra i vari processori.
Ciò detto, si può indicare con $T_p(A)$ il tempo dell'algoritmo parallelo e per esso si ha:

$$T_p(A) = T_{seq}(A) + \frac{T_{par}(A)}{p}, \text{ dove } T(A) = T_1(A) = T_p(A), \text{ per p=1} \tag{1.13}$$

Definizione 1.25 (Speed-Up) *se $T(A)$ è il tempo di esecuzione di un algoritmo su 1 processore e $T_p(A)$ è il tempo di esecuzione dell' algoritmo parallelo su p processori [4, p. 157] . Lo speed up S_p è definito così:*

$$S_p = S(p) = \frac{T(A)}{T_p(A)} < p, \ \forall \ p > 1 \tag{1.14}$$

Lo speed-up misura il beneficio ottenuto dall'introduzione del parallelismo. La funzione:

$$S(p) = p \tag{1.15}$$

è detta speed-up ideale.

Definizione 1.26 (Overhead Totale) *Si indica con overhead totale O_h la misura di quanto lo speed-up differisce da quello ideale.*

$$O_h = p \cdot T_p(A) - T(A) \tag{1.16}$$

$$T_p(A) = \frac{(O_h + T(A))}{p}$$

$$S_p = \frac{T(A)}{T_p(A)} = \frac{T(A)}{\frac{(O_h+T(A))}{p}} = \frac{p \cdot T(A)}{O_h + T(A)} = \frac{p}{\frac{O_h}{T(A)} + 1}$$

Definizione 1.27 (Efficienza) *Sia p il numero di processori e S_p lo speed-up relativo ad esso. Si definisce efficienza E_p il parametro*

$$E_p = E(p) = \frac{S_p}{p} \tag{1.17}$$

che fornisce un'indicazione di quanto sia stato utilizzato il parallelismo del calcolatore [4, p. 159]. La funzione

$$E(p) = 1 \tag{1.18}$$

è detta efficienza ideale.

Dalla definizione di efficienza (1.27) ed overhead totale (1.26):

$$E_p = \frac{S_p}{p} = \frac{T(A)}{p \cdot T_p(A)} = \frac{T(A)}{O_h + T(A)} = \frac{1}{\frac{O_h}{T(A)} + 1}$$

Definizione 1.28 (Legge di Amdahl) *Si ha che [10]:*

$$S_p = \frac{1}{\alpha + \frac{1-\alpha}{p}} \tag{1.19}$$

dove:

$$\alpha = \frac{T_{seq}(A)}{T(A)} \tag{1.20}$$

infatti:

$$S_p = \frac{T(A)}{T_p(A)} = \frac{T_{seq} + T_{par}}{T_{seq} + \frac{T_{par}}{p}} = \frac{1}{\frac{T_{seq}}{T(A)} + \frac{T_{par}}{T(A) \cdot p}} = \frac{1}{\frac{T_{seq}}{T(A)} + \frac{T_{par}}{p}} \overset{1.20}{=} \frac{1}{\alpha + \frac{1-\alpha}{p}}$$

Assumendo α costante rispetto a p poiché:

$$\lim_{p\to\infty} \frac{1-\alpha}{p} = 0$$

risulta quindi:

$$S_p \leq \frac{1}{\alpha}$$

Ciò significa che la parte sequenziale può degradare fortemente lo speed-up quando il numero di processori è sufficientemente elevato [4].
Fissato un algoritmo A e la sua implementazione su una macchina dotata di p processori si definirà il:

Definizione 1.29 (Tempo del software parallelo) *Detto $T_{sw}(A)$ il tempo del software che implementa l'algoritmo A definito in 1.4, considerando la sua implementazione in una macchina multiprocessore bisogno tener conto anche dei tempi di sincronizzazione, comunicazione, etc... per cui si ha:*

$$T_{sw,p}(A) = T_{seq} + \frac{T_{par}}{p} + T_O(p) \tag{1.21}$$

dove in T_O sono compresi i tempi di comunicazione, sincronizzazione, etc.

Ciò vuol dire che nella valutazione di software paralleli si deve tener conto anche di tali fattori oltre che al *calcolo* e agli accessi alla *memoria*. In questo contesto si considereranno i tempi di comunicazione e di sincronizzazione come il livello più alto di accesso alla memoria.
E' importante nella valutazione delle prestazioni di un software parallelo il concetto di scalabilità ovvero la stima delle prestazioni al variare del numero dei processori [11].
Si distinguono due tipologie di scalabilità:

Definizione 1.30 (Strong Scalability) *La dimensione del problema resta fissata e cresce il numero di unità processanti [12].*

Al variare del numero di processori si studia il comportamento dello speed-up (Figura 1.6) che idealmente dovrebbe essere uguale a p.

Definizione 1.31 (Weak Scalability) *La dimensione del problema cresce con il numero di unità processanti, mantenendo lo stesso carico di lavoro per unità processante. [12].*

Al variare del numero di processori, assegnando ad ogni processore la stessa dimensione del problema, si osserva l'andamento del tempo di esecuzione che idealmente dovrebbe restare costante (Figura 1.7).

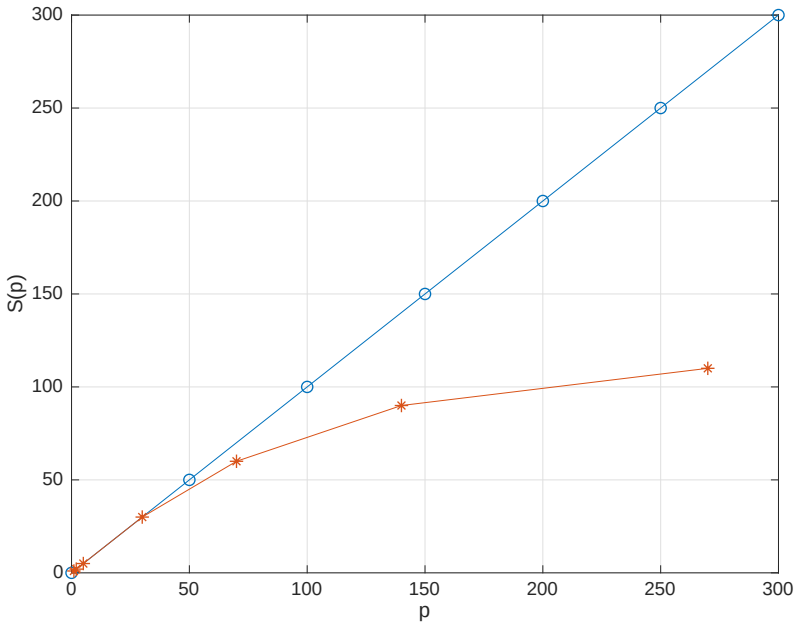


Figura 1.6: Studio della strong scalability, al crescere del numero di processori lo speed-up effettivo degrada mentre lo speed-up ideale cresce linearmente in funzione di p

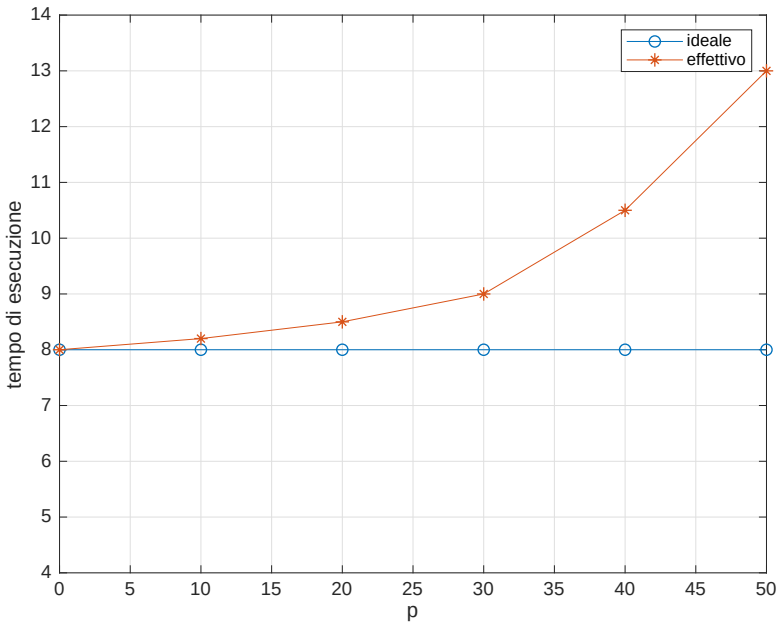


Figura 1.7: Weak scalability: al crescere del numero di processori p, dove ciascun processore risolve un problema di ugual dimensione, il tempo idealmente dovrebbe restare costante

1.2 Architetture di riferimento

In questa sezione si illustra come è stata determinata la prestazione di picco degli ambienti di calcolo utilizzati. In tabella 1.4 si possono osservare le caratteristiche dell'ambiente desktop della Intel.

1.2.1 Computer cluster - Mont-Blanc [ARK1]

Il Mont-Blanc dispone di 930 nodi, e ciascun nodo è composto da un processore *ARM Cortex A15* (1.2), generalmente usati per smartphone e tablet, tali processori hanno spesso prestazioni basse poiché prediligono i bassi consumi, ma rappresentano comunque un buon compromesso dove poter utilizzare il calcolo parallelo e distribuito. La prestazione di picco del nodo è stata così calcolata:

$$2(cores/node) * 2(Flops * core) * 2.6(GHz) = 6.8(GFLOPs/node) \quad (1.22)$$

Per la (1.22), si può affermare che l'intero cluster raggiunge i 6324 TFLOPs di picco delle prestazioni.

Mont-blanc prototype	
Ogni nodo ha	SoC Samsung Exynos 5 Dual CPU ARM 2x Cortex-A15 @ 1.7GHz
RAM	4GB DDR3

Tabella 1.2: Caratteristiche e prestazione di picco del cluster Mont-Blanc del BSC

Mont-blanc prototype	
Ogni nodo ha	SoC Samsung Exynos 5 Dual CPU ARM 2x Cortex-A15 @ 1.7GHz
Prestazione di picco per ciascun core	3.4 GFLOPs
Prestazione di picco per ciascun processore	6.8 GFLOPs
Prestazione di picco per l'intero cluster	6324 GFLOPs

Tabella 1.3: Caratteristiche e prestazione di picco del cluster Mont-Blanc del BSC

1.2.2 Desktop computer [ARK2]

Si è deciso di utilizzare un computer desktop per rendere più facile il calcolo manuale della prestazione sostenuta. Per cui i test del paragrafo A.0.1 sono

stati eseguiti su Intel Core^{TM} i5-3470S CPU @ 2.90GHz che ha queste caratteristiche La prestazione di picco dichiarata dal produttore del processore Intel i5-3470s è la seguente:

Processor Number	*Frequency Type*	*Clock (GHz)*	*GFLOPs*
i5-3470S	Base	2.9	92.8
	Memoria	8 Gbyte DDR3	
	Core	4	

Tabella 1.4: Architettura utilizzata per i test

La prestazione di picco è stata calcolata:.

$$4(core) * 8(FLOP/cycle) * 2.9GHz = 92.8(GFLOPs) \tag{1.23}$$

il numero di *FLOP per ciclo di clock* può essere ottenuto andando ad approfondire i manuali dell'architettura rilasciati dal produttore, in questo caso è stato consultato il manuale relativo alla famiglia di processori Sandy Bridge [13].

Capitolo 2

Strumenti per valutare le prestazioni della macchina

2.1 Benchmark

Un benchmark è un programma che serve ad ottenere la prestazione di macchine mediante un insieme di test (software) che hanno l'obiettivo di creare una metrica univoca per classificarle [14].
Qui di seguito saranno presentanti alcuni dei benchmark maggiormente utilizzati dai sistemi di calcolo che identificano una applicazione di riferimento con cui confrontare l'applicazione sviluppata.:

- Linpack Benchmark
- HPCG
- STREAM

2.1.1 Linpack benchmark

Il benchmark Linpack è stato introdotto dal gruppo di ricerca di Jack Dongarra [15]. Benchè la libreria Linpack si sia successivamente evoluta nella libreria LAPACK con l'evoluzione delle nuove architetture, il benchmark linpack resta comunque un punto di riferimento per la misura delle prestazioni dei calcolatori. Il benchmark LINPACK risolve un sistema di equazioni lineari denso poiché è stato stimato che il calcolo della soluzione di un sistema lineare costituisce il nucleo principale del processo di risoluzione di almeno il 70% di tutti i problemi scientifici [16]. Il benchmark LINPACK offre due possibili modalità d'esecuzione. Una in cui si risolve un sistema

di equazioni lineari di ordine 100 (100 equazioni) e si esegue il software senza alcuna particolare configurazione, l'altra in cui si risolve un sistema di dimensione 1000 effettuando tutte le configurazioni necessarie (tuning) affinchè il software ottenga le massime prestazioni[7], da notare che la misura della prestazione ottenuta non riflette la prestazione complessiva del sistema, ma riflette piuttosto la prestazione di un sistema dedicato a risolvere sistemi di equazioni lineari, i risultati forniti offrono una buona stima delle prestazioni di picco. I programmi LINPACK sono caratterizzati da un'alta percentuale di operazioni aritmetiche floating-point, questo come esposto precedentemente è uno dei motivi che porta a tenerlo in considerazione come misura. Le routine matematiche coinvolte in questo studio, SGEFA e SGESL (la prima fattorizza la prima A mediante il metodo di eliminazione Gaussiana, la seconda risolve il sistema lineare $Ax = b$ o $A^T x = b$ usando i fattori calcolati dalle routine SGECO o SGEFA [17]) utilizzano algoritmi memorizzati per colonne (column-major), ci si riferisce ad array di elementi sequenziali memorizzati per colonna e non per riga,l'orientamento a colonna è importante per l'aumento dell'efficienza poiché è il modo in cui Fortran memoriazza i vettori, questa può essere una delle ottimizzazioni di cui si è parlato precedentemente. In LINPACK la maggior parte delle operazioni in virgola mobile è svolta da un insieme di sottoprogrammi detti, Basic Linear Algebra Subprograms (BLAS), letteralmente Sottoprogrammi per l'Algebra Lineare di Base [18], richiamati ripetutamente durante il calcolo. L'utilizzo di routine della libreria BLAS, ottimizzata per calcoli matriciali e vettoriali, è un altro dei motivi che spinge la comunità scientifica a ritenere ancora valida questa misura. Le routine BLAS fanno riferimento a quelle del livello 1, in cui vengono utilizzati array monodimensionali e non array bidimensionali.

I compilatori di alcune macchine possono, ovviamente, generare codice ottimizzato che in maniera autonoma accede alle caratteristiche speciali della macchina, nel caso di sistemi con 100 equazioni però non vengono utilizzate ottimizzazioni di questo tipo (Figura 2.1). L'arrivo dei computer altamente paralleli c'è la necessità di avere dei benchmark su problemi che abbiano un senso, quindi avere matrici di dimensioni 100×100 o 1000×1000 è poco utile . Per questo motivo è nata la versione scalabile di LINPACK Benchmark, chiamata HPL [15].

Un vincolo importante comunque è che tutti programmi ottimizzati mantengano la stessa accuratezza relativa delle tecniche standard, come quelle dell'eliminazione gaussiana usata in LINPACK. Il primo passo consiste nell'esecuzione del driver (fornito con LINPACK Benchmark) per assicurarsi che venga risolto lo stesso problema. Il driver inizializza le matrici, chiama

le funzioni per risolvere il problema, verifica che la risposta sia corretta e calcola il numero totale di operazioni per la risoluzione (indipendentemente dal metodo) eseguendo $\frac{2 \cdot n^3}{3} + 2n^2$.
L'ultima colonna della Figura 2.1 è la prestazione di picco teorica fornita dal produttore che garantisce che il programma non ecceda mai questo limite, una sorta di velocità della luce per un calcolatore.
La prestazione di picco teorica è determinata calcolando il numero di operazioni in virgola-mobile, addizioni e moltiplicazioni, nella massima precisione fornita dalla macchina, che possono essere svolte durante un periodo di tempo, tipicamente il ciclo di clock della macchina, come osservato in 1.2.

Esempio 2.1 *Ad esempio il Cray Y-MP/8 ha una durata del ciclo di 6 ns. Durante il ciclo di clock i risultati di una addizione e di una moltiplicazione possono essere ottenuti in*

$$\frac{2 \ N_c}{1 \ clock \ cycle} \times \frac{1}{6 \ ns} \tag{2.1}$$

dove la frequenza è pari a $\frac{1}{6 \ ns}$ *sul singolo processore. Il Cray ha 8 processori per questo motivo la prestazione di picco teorica è di* 2.6 *GFLOPs (1.2).*

La colonna etichettata con **Computer** fornisce il nome del calcolatore sul quale il programma è stato eseguito. In alcuni casi si indica il numero di processori della configurazione e in altri casi la durata del ciclo di clock del processore in nanosecondi. La colonna etichettata con **LINPACK benchmark** indica il sistema operativo e il compilatore usato. La colonna etichettata con TPP (Toward Peak Performance - Verso la prestazione di picco) fa riferimento ai risultati dell'ottimizzazione effettuata dal programmatore; la dimensione del problema è 1000. La colonna finale etichettata con **Theoretical Peak** dà la massima velocità di esecuzione basata sulla durata del ciclo di clock dell'hardware, utilizzando la stessa matrice per risolvere un sistema di equazioni. I risultati dell'accuratezza vengono confrontati calcolando il residuo del problema

$$\frac{||Ax - b||}{(||A||||x||)}$$

Il residuo deve essere inferiore a $n\epsilon$ dove n indica l'ordine della matrice ed ϵ è la precisione della macchina, ovvero 2^{-53}, su un computer con standard IEEE. Le informazioni presenti in Figura 2.1 sono state raccolte durante un certo periodo di tempo, infatti il documento di contenente tali informazioni viene pubblicato ogni 10 anni [7].

Computer	"LINPACK Benchmark" OS/Compiler	n=100 Mflop/s	"TPP" Best Effort n=1000 Mflop/s	"Theoritical Peak" Mflop/s
Intel Pentium Woodcrest (1 core, 3 GHz)	ifort -parallel -xT -O3 -ipo -mP2OPT_hlo_loop_unroll_factor=2	3018	6542	12000
Intel Pentium Woodcrest (1 core, 2.67 GHz)	ifort -O3 -ipo -xT -r8 -i8	2636		10680
Intel Core 2 Q6600 Kensfield) (4 core, 2.4 GHz)			13130	38400
Intel Core 2 Q6600 Kensfield) (3 core, 2.4 GHz)			11980	28800
Intel Core 2 Q6600 Kensfield) (2 core, 2.4 GHz)			9669	19200
Intel Core 2 Q6600 Kensfield) (1 core, 2.4 GHz)	ifort -O3 -xT -ipo -static -i8 -mP2OPT_hlo_loop_unroll_factor=2	2426	7519	9600
NEC SX-8/8 (8proc. 2 GHz)			75140	128000
NEC SX-8/4 (4proc. 2 GHz)			43690	64000
NEC SX-8/2 (2proc. 2 GHz)			25060	32000
NEC SX-8/1 (1proc. 2 GHz)	-pi -Wf"-prob_use"	2177	14960	16000
HCL Infiniti Global Line 4700 HW (4 proc Intel Xeon 3.16 GHz)	ifort -fast -r8 -align	1892	9917	25280
HP ProLiant BL20p G3 (2 proc (1 cpu core per single chip), 3.8GHz Intel Xeon)			8185	14800
HP ProLiant BL20p G3 (1 proc (1 cpu core per single chip), 3.8GHz Intel Xeon)	SuSE SLES 9 / Intel 8.1 Compile flags: -fpp -xP -O3 -openmp -align -ipo	1852	4851	7400
HP ProLiant DL360 G4 (2 proc, 3.6GHz/1MB Xeon)			7031	14400
HP ProLiant DL360 G4 (1 proc, 3.6GHz/1MB Xeon)	Intel 8.1 -fpp -xW -O2 -unroll -align -openmp	1821	4220	7200
HP ProLiant DL360 G4p (2 proc (1 cpu core per single chip), 3.8GHz Intel Xeon)			8155	14800
HP ProLiant DL360 G4p (1 proc (1 cpu core per single chip), 3.8GHz Intel Xeon)	SuSE SLES 9 / Intel 8.1 Compile flags: -fpp -xP -O3 -openmp -align -ipo	1861	4860	7400
HP ProLiant DL140 G2 (2 proc (1 cpu core per single chip), 3.8GHz Intel Xeon)			8163	14800
HP ProLiant DL140 G2 (1 proc (1 cpu core per single chip), 3.8GHz Intel Xeon)	SuSE SLES 9 / Intel 8.1 Compile flags: -fpp -xP -O3 -openmp -align -ipo	1861	4858	7400
HP ProLiant ML370 G4 (2 proc (1 cpu core per single chip), 3.8GHz Intel Xeon)			8111	14800
HP ProLiant ML370 G4 (1 proc (1 cpu core per single chip), 3.8GHz Intel Xeon)	SuSE SLES 9 / Intel 8.1 Compile flags: -fpp -xP -O3 -openmp -align -ipo	1851	4835	7400
HP ProLiant DL380 G4 (2 proc (1 cpu core per single chip), 3.8GHz Intel Xeon)			8198	14800
HP ProLiant DL380 G4 (1 proc (1 cpu core per single chip), 3.8GHz Intel Xeon)	SuSE SLES 9 / Intel 8.1 Compile flags: -fpp -xP -O3 -openmp -align -ipo	1851	4882	7400
Intel Pentium Nocona 3.6 GHz	ifort -O3 -xP -ipo -align -r8	1803	3385	7200
Intel xeon 64 (dual) 3.6 GHz	ifort -fast -r8 -align.	1779	7278	14400
IBM eServer p5 575 (8 proc, 1.9 GHz POWER5)			34570	60800
IBM eServer p5 575 (1 proc, 1.9 GHz POWER5)	-O3 -qarch=pwr5 -qtune=pwr5 -Pv -Wp,-ea478,-g1	1776	5872	7600

Figura 2.1: Un esempio di classificazione di calcolatori creata da Jack Dongarra

2.1.2 Risultati di Linpack Benchmark su [ARK1]

In tabella 2.1 ci sono gli output ottenuti su un singolo nodo (usando 1 solo core) per il cluster Mont-Blanc usando Linpack Benchmark. La prestazione

Dimensione sistema	*GFLOPs*	*Pres. Picco (GFLOPs)*	Core	% picco
100	0,190355	3.4	1	5,5%
200	0,195794	3.4	1	5,5%
300	0,203151	3.4	1	5,9%
400	0,202378	3.4	1	5,9%
500	0,206737	3.4	1	6,05%
1000	0,167900	3.4	1	4,7%

Tabella 2.1: Risultati su Mont-Blanc con Linpack Benchmark

sostenuta ottenuta è di circa 0.2 GFLOPs per la risoluzione di un sistema di equazioni lineari di dimensione 100 su un singolo nodo. Si nota che all'aumentare della dimensione del sistema, dalla dimensione 100 fino ad arrivare ad un sistema di dimensione $n = 1000$ il cluster raggiunge solo lo 6.05% della prestazione di picco su un processore (considerando la 1.22).

2.1.3 HPL

Per confrontare le macchine parallele è stata realizzata una versione di Linpack Benchmark parallela chiamata *HPL*, un'implementazione portabile e scalabile del benchamark LINPACK scritta in fortran per sistemi distribuiti [19], in maniera tale da poter scalare la dimensione del problema a piacimento e ripetere lo stesso benchmark su calcolatori paralleli, utilizzando la risoluzione di sistemi di equazioni lineari densi. Per gli esperimenti riportati (Figura 2.1) sono stati usati gli algoritmi standard LINPACK, o algoritmi basati su tecniche matrice-matrice mentre per gli esperimenti in Figura 2.2 è stato utilizzato il benchmark HPL. L'algoritmo usato da HPL utilizza la fattorizzazione LU con il pivoting parziale, utilizzando tecniche definite il letteratura come 'look-ahead' o 'compute-ahead'. In particolare, le operazioni floating-point in doppia precisione dell'algoritmo sono $\frac{2}{3}n^3 + O(n^2)$. I dati vengono distribuiti in una griglia bidimensionale di processori $P \times Q$ in maniera ciclica a blocchi per assicurare che il carico di lavoro sia ben bilanciato e per aumentare la scalabilità dell'algoritmo. La matrice $n \times n$ è partizionata in $NB \times NB$ blocchi distribuiti nella griglia $P \times Q$ in entrambe le direzioni della matrice [20]. Di seguito alcuni dei parametri utilizzati per classificare i sistemi di calcolo paralleli (Figura 2.2)

- #Proc - Numero di processori (core)

Computer (Full Precision)	Number of Procs or Cores	R_{max} GFlop/s	N_{max} Order	$N_{1/2}$ Order	R_{Peak} GFlop/s
NUDT, Inspur Tianhe-2 (TH-2) Model TH-IVB-FEP Nodes=16000 2 Intel Xeon IvyBridge (6 core) E5-2692 2.2GHz & 3 Intel Xeon Phi 31S1P	2371200	22808300	6974976		41733120
IBM Blue Gene/Q Power BCQ 1.6 GHz (120 racks * 1024 nodes/rack * 16 cores/node) w/Custom	1966080	21466530	14942207		25165824
IBM Blue Gene/Q Power BCQ 1.6 GHz (96 racks * 1024 nodes/rack * 16 cores/node) w/custom	1572864	16324751	12681215		20132659
IBM Blue Gene/Q Power BCQ 1.6 GHz (72 racks * 1024 nodes/rack * 16 cores/node) w/custom	1179648	12003644	10715135		15099494
K computer, Fujitsu SPARC64 VIIIfx 2.0GHz, 8 core w/Tofu interconnect	705024	10510000	11870208		11280384
K computer, Fujitsu SPARC64 VIIIfx 2.0GHz, 8 core w/Tofu interconnect	548352	8162000	10725120		8773632
IBM Blue Gene/Q Power BCQ 1.6 GHz (48 racks * 1024 nodes/rack * 16 cores/node) w/custom	786432	8152590	8912895		10066330
IBM Blue Gene/Q Power BCQ 1.6 GHz (24 racks * 1024 nodes/rack * 16 cores/node) w/custom	393216	4141180	6422527		5033165
IBM iDataPlex dx360 M4 2 x Intel E5-2680v2 (2.8 GHz) Ivy Bridge CPU Cores: 26,100 (1305 nodes * 2 sockets * 10 cores/socket) GPUs: NVIDIA 2 x K20x -GPU cores: 36,540 InfiniBand FDR	62640	3003000	3838464		4003740
IBM iDataPlex DX360M4 Intel Sandybridge 2.7 GHz (9216 nodes * 2 sockets * 8 cores/socket) w/InfiniBand	147456	2897000	5201920		3185050
TH-1A (14336 6-core Intel X5670 2.93 GHz + 7168 Nvidia M2050 w/custom interconnect)	186368	2566000	3600000	1000000	4701061
IBM iDataPlex DX360M4 Intel Sandybridge 2.7 GHz (7168 nodes * 2 sockets * 8 cores/socket) w/InfiniBand	114688	2072000	4464640		2477261
IBM Power 775 (IBM POWER7 3.836 GHz w/Custom (equivlent to 247.5 drawers x 8 sockets per drawer x 32 cores per socket))	63360	1515000	2280960		1944392
IBM Power 775 (IBM POWER7 3.836 GHz) (216 drawers x 8 sockets per drawer x 32 cores per socket) Custom	55296	1429000	4147200		1696923
IBM Blue Gene/Q Power BCQ 1.6 GHz (8 racks * 1024 nodes/rack * 16 cores/node) w/custom	131072	1358197	3899391		1677721
464 Dawning TC3600 Blade System 4640 Computing Nodes (2*Intel 6 core X5650 2.666 GHz, 1*NVidia Tesla C2050 GPU) w/InfiniBand	120640	1271000	2359296		2983520
IBM Power 775 (POWER7 3.836 GHz, w/custom) (8 sockets per drawer x 32 cores per socket)	47488	1183000	3419136		1457311
IBM BladeCenter cluster of 3240 nodes dual socket 1.8 GHz Opteron (dual core) LS21 blades plus 6480 nodes dual socket 3.2 GHz PowerXCell 8i (8 SPU + 1 PPU cores) QS22 blades w/InfiniBand	129600	1105000	2329599		1456704
TSUBAME 2.0; 1357 HP Proliant SL390s G7 nodes w/ Xeon X5670 (2.93GHz) 6cores x 2sockets, NVIDIA Tesla M2050 (1.15GHz) 14cores x 3chips and QDR InfiniBand x 2rails; SUSE Linux Enterprise server 11	73278	1192000	2490368		2287630
Cray XT5 (Opteron quad core 2.3 GHz)	150152	1059000	4712799		1381400
Cray XE6 (AMD 12-core, 2.1Ghz w/custom interconnect)	153408	1054000	4537344		1288627
IBM BladeCenter cluster of 3060 nodes dual socket 1.8 GHz	122400	1042000	2249343		1375776

Figura 2.2: Risultati ottenuti dal benchmark HPL su alcune architetture d'esempio

- Rmax - Prestazione Linpack massima raggiunta (prestazione sostenuta da linpack)
- Rpeak - Performance di picco teorica
- Nmax - Dimensione del problema per raggiungere Rmax
- N1/2 - Dimensione del problema per raggiungere la metà di Rmax

2.1.4 Risultati di HPL [ARK1]

Ha senso capire la prestazione sostenuta da Linpack Benchmark nella sua versione distribuita (HPL). Nel paragrafo di precedente si sono mostrati i risultati ottenuti con Linpack Benchmark utilizzando la versione seriale del software per capire quale fosse la prestazione di un solo nodo. I risultati ottenuti sono stati testati utilizzando 128 nodi, sebbene il Mont-Blanc ne metta a disposizione molti di più, dato che il totale dei nodi dichiarato si trova su code diverse per questo motivo non è stato possibile utilizzarle tutte contemporaneamente, quindi si è deciso di utilizzare una dimensione quando più vicina ad una potenza del due. Per essere mandato in esecuzione HPL ha bisogno di un file di configurazione in input (chiamato generalmente *HPL.dat*), i valori necessari sono:

- P,Q, dimensione della griglia da assegnare a ciascun processore. I produttori consigliano di sceglierla quanto più quadrata possibile, e ove necessario con Q leggermente maggiore in dimensione;
- Ns, la dimensione del problema;
- NB, la dimensione del blocco nella griglia.

La dimensione della griglia viene calcolato sommariamente in base al:

- Dimensione della memoria a disposizione
- Dimensione del tipo dei dati (generalmente DOUBLE)

Quindi nel caso del Mont-Blanc un calcolo approssimativo considerando di avere 128 nodi a disposizione e 2,8 GByte di memoria ram (realmente usabili):

$$Ns = \sqrt{(2.800.000.000Byte * 128)/8 * 0,7} = 177000$$

quindi la dimensione del problema dovrebbe essere di circa 177.000 considerando di voler occupare circa il 70% della memoria. I risulati (tabella 2.2)

mostrano che variando la dimensione del problema (fino al massimo consentito dal sistema) il valore della prestazione sostenuta si attesta al 66% circa confrontandola alla prestazione di picco di $435 GFLOPs$.

# processori	*Ns*	$P \times Q$	*Sostenuta GFLOPs*	*Picco GFLOPs*	*% picco*
128	20.000	8×16	123,4	435,2	28%
128	50.000	8×16	228,7	435,2	52%
128	60.000	8×16	245,2	435,2	56%
128	100.000	8×16	286,8	435,2	66%

Tabella 2.2: Dati ottenuti con l'esecuzione di HPL sul Mont-Blanc

2.1.5 TOP500

Il motivo per cui si è parlato di Linpack Benchmark è quello di illustrare anche i contesti in cui viene utilizzato, uno di questi è il progetto TOP500 che mira a creare una lista aggiornata dei 500 più potenti supercomputer del pianeta. Il progetto venne avviato nel 1993 e aggiorna la lista dei sistemi due volte all'anno. Il primo aggiornamento coincide con l'International Supercomputing Conference di giugno, il secondo si ha a novembre durante la ACM/IEEE Supercomputing Conference. Il progetto mira a fornire una base di riferimento comune per analizzare l'evoluzione del supercalcolo. Le prestazioni dei sistemi vengono valutate tramite il benchmark HPL.
Sul sito web `www.top500.org` vengono riportati alcuni indici di classificazione il cui significato è:

- NWORLD: Posizione nella TOP500
- Manufacturer: Produttore o venditore
- Installation Site: Nome del proprietario del sistema di calcolo
- Location: Posizione e paese
- Year: Anno di installazione o ultimo aggiornamento importante

Vengono analizzati i pro ed i contro nell'utilizzo di un benchmark come Linpack:
Pro

- Una sola metrica per giudicare la qualità;

Rank	*System*	*Cores*	*Rmax (TFLOPs)*	*Rpeak (TFLOPs)*	*Power (kw)*
1	Sunway TaihuLight	10,649,600	93,014.6	125,435.9	15,371
2	Tianhe-2	3,120,000	33,862.7	54,902.4	17,808
3	Piz Daint	361,760	19,590.0	25,326.3	2,272
4	Gyoukou	19,860,000	19,135.8	28,192.0	1,350
5	Titan	560,640	17,590.0	27,112.5	8,209

Tabella 2.3: Le prime posizione della TOP500 aggiornate a Novembre 2017

- Semplice da definire e indicizzare;
- Permette alla dimensione del problema di cambiare con il calcolatore nel tempo;
- Permette la competizione tra diversi sistemi di calcolo.

Contro

- Si focalizza solo la prestazione di picco della CPU;
- Non "stressa" il bandwidth locale;
- Non "stressa" la rete;
- Un solo indice non può riflettere le prestazioni dell'intero sistema.

2.1.6 HPCG

Per superare alcuni dei limiti di Linpack Benchmark è stato introdotto il benchmark *High Performance Conjugate Gradient*(HPCG) [21],[22]. Il metodo operato da HPCG è quello del gradiente coniugato per risolvere un sistema di equazioni lineari, e risolve sistemi sia densi, con una alta intensità computazionale (definita in 3.5), che sparsi. Il motivo per cui viene introdotto HPCG è perché è cresciuto l'uso e la pervasività dei metodi iterativi che risolvono problemi tecnico scientifici su larga scala.
Gli obiettivi di HPCG sono molteplici:

- Fornire una copertura dei pattern di comunicazioni più comuni (ad es. effettuare benchmark per gli scambi tra domini di dati vicini, o prodotti scalari e moltiplicazioni di matrici e vettori in parallelo);

- Investire nel miglioramento di funzioni di comunicazione collettive che attualmente sono tra i maggiori colli di bottiglia delle applicazioni parallele, quindi migliorare le performance del benchmark stesso implica un miglioramento in termini di performance delle applicazioni reali;

Lo scopo di ogni benchmark, e di HPCG in particolare, è trovare un compromesso tra complessità, dipendenza del software, e requisiti imposti dalle reali applicazioni. HPCG è scritto in *C++* per utilizzare le funzioni dei compilatori di ultima generazione affinchè il benchmark sfutti le caratteristiche più comuni delle moderne architetture, inoltre è stato sviluppato anche nell'ottica di utilizzare gli acceleratori come GPU e coprocessori.

In tabella 2.4 si possono osservare i risultati ottenuti con il benchmark aggiornati a Novembre 2015. E' interessante notare la differenza che la classifica fornisce tra HPL e HPCG, si noti che HPL ottiene valori molto più alti di prestazione rispetto ad HPCG questo è in parte spiegabile con quello che spesso viene definito *memory wall*, ovvero il limite di bandwidth imposto dalla memoria e su cui hpcg si basa. In pratica HPCG raggiunge una fra-

Rank	*System*	*Cores*	*HPL Rmax (PFLOPs)*	*TOP500*	*HPCG (PFLOPs)*	*% picco*
2	Tianhe-2	3,120,000	33.863	1	0.5800	5.3%
1	K-Computer	705,024	10.510	4	0.4608	1.1%
3	Titan Cray	560640	17.590	2	0.3223	1.2%
4	Trinity	301056	8.101	6	0.1826	1.6%
5	Mira	786432	8.587	5	0.1670	1.7%

Tabella 2.4: Le prime posizione della classifica dell'HPCG aggiornate a Novembre 2015

zione molto bassa della prestazione di picco, che è maggiormente legata al numero di FPU, il throughput, il mix di istruzioni utilizzabili come pipeline, addizione, moltiplicazione e FMA. Solo LINPACK benchmark può ottenere sufficiente riuso di dati e località nei suoi modelli computazionali in modo da essere adatto per mantenere quante più FPU occupate ad ogni ciclo, e così ottenere una buona percentuale della prestazione di picco teorica teorica dell'hardware. I risultati di performance di HPCG sono largamente dipendenti dalla gerarchia di memoria e dalla qualità delle connessioni, il bandwidth, la latenza, e il parallelismo nel trasferimento dei dati tutte contribuiscono alla performance totale riportata in HPCG.
HPCG è un benchmark molto giovane per essere paragonato a LINPACK

Benchamrk, e negli anni quando la misura si stabilizzerà comunque si potranno avere risultati molto diversi da quelli mostrati dovuti a massicci cambi di codice sorgente, e potrebbero anche esserci risultati molto differenti tra un anno e l'altro.
Un'altra interessante metrica per aiutare a comparare i sistemi è quella di contare il numero di inversioni tra la TOP500 e la lista generata da HPCG (in tabella 2.4 la colonna *TOP500*). Un importante esempio l'inversione che avviene tra il Titan supercomputer e il K-computer. Mentre il primo è davanti nella TOP500 quest'ultimo ha un migliore posizionamento nella lista HPCG. Vale la pena notare che entrambi i sistemi hanno una versione ottimizzata da esperti di HPL e HPCG, che garantisce il raggiungimento dei massimi risultati di performance. La prestazione di picco relativa a HPL del Titan è di 27 PFLOPs che è superiore a quella del K-computers (11 PFLOPs) probabilmente dovuto all'uso delle GPU NVIDIA per cui si avvicina maggiormente alla prestazione di picco. Comunque, Titan raggiunge solo il 65% del picco mentre il K-computer raggiunge il 93%. La ragione di questa differenza così marcata è dovuta alle interconnessioni delle due macchine. K-computer usa le interconnessioni TOFU [23] mentre Titan usa Cray Gemini [24]. La prima è altamente sovradimensionata in relazione alle performance mentre l'ultima resta superiore dal punto di vista commerciale ed ottiene un rapporto $\frac{byte}{flops}$ piccolo. Ciò lascia intuire che quando si lancia in esecuzione HPCG, l'importanza di una rete veloce cresce a causa del contributo dovuto ai tempi di comunicazione delle routine collettive e gli scambi sui bordi. Osservando i risultati, K-computer raggiungere 0,46 PFLOPs al secondo e Titan 0,32 PFLOPs al secondo. Quindi il K-computer arriva ad una frazione di picco del 4% mentre il Titan arriva all'1%.
In linea di massima il benchmark HPCG divide i supercomputer in tre grandi insiemi:

- I sistemi vettoriali con una un bandwidth di memoria molto alto raggiungono frazioni delle prestazioni di picco intorno al 10% (Earth Simulator);
- Le macchine molto specializzate con buone interconnessioni raggiungono il 5% della prestazione di picco (K-Computer);
- I sistemi più comuni si fermano ad un valore dell'1% (Titan, Tianje-2).

Questo esprime ciò succede anche nella TOP500 dove sistemi con interconnessione *Infiniband* raggiungono il 70% della prestazione di picco, mentre quelli con *Ethernet* il 30%.

2.1.7 Risultati di HPCG [ARK1]

Di seguito si osservano i risultati ottenuti con HPCG Benchmark. Come è

# processori	*Tempo impiegato*	*Dimensione griglia*	*Pres. Sost (GFLOPs)*	*Pres. Picco (GFLOPs)*	*% picco*
8	107.237	$208 \times 208 \times 208$	1.51645	27.2	5.5%
64	112.599	$416 \times 416 \times 416$	11.5746	217,6	5.3%
128	114.67	$832 \times 416 \times 416$	23.4612	435.2	5.3%

Tabella 2.5: Risultati ottenuti con HPCG

evidente dai risultati ottenuti da HPCG anche se con applicazioni di benchmarking che sfruttano in maniera intensiva le unità dedicate al calcolo delle operazioni floating point si ottiene soltanto il 5.5% circa delle prestazioni di picco raggiungibili dal sistema di calcolo. Anche in questo caso si ottiene una prestazione ben al di sotto del picco di prestazione anche utilizzando più processori ma in linea con quelle ottenute dalla lista riferita ai maggiori supercomputer del mondo 2.4.

2.2 Altri benchmark

2.2.1 STREAM

Dalle considerazioni nate da HPCG si intuisce che monitorare le prestazioni delle memorie locali è un argomento centrale per le applicazioni. Per questo effettuare benchmark anche sul bandwidth di memoria è fondamentale. Un benchmark molto utilizzato con questa finalità è STREAM (Figura 2.3), creato da John D. McCalpin ricercatore al TACC (Texas Advanced Computing Center).
Un dato che spinge alla riflessione è che la velocità della CPU sta crescendo molto rapidamente rispetto alla velocità del sistema di memoria della macchina. Molti software attualmente sono limitati dall'uso efficiente del bandwidth di memoria del sistema piuttosto che dalla potenza del sistema di calcolo [25]. L'autore di STREAM per rendere chiara la situazione dice "facendo un esempio estremo, moltissimi sistemi di calcolo di alta fascia eseguono semplici operazioni aritmetiche sfruttando al 4% o 5% le loro prestazioni di picco. Questo vuol dire che consumano la maggior parte del tempo aspettando il completamento di cache miss" quindi hanno un alto traffico parassita (definito in 1.7)) ed effettuano maggiori accessi alla memo-

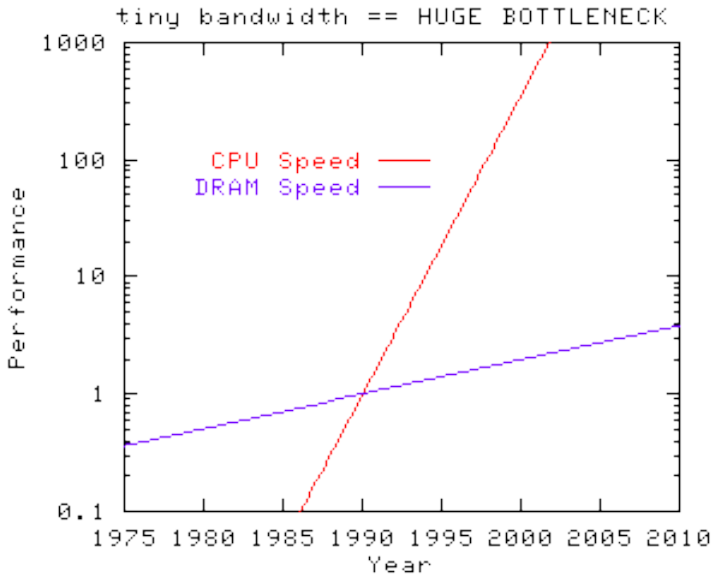


Figura 2.3: Il logo di STREAM mostra la crescita negli anni della veloità delle CPU rispetto alla velocità della memoria di un sistema

ria rispetto alle operazioni eseguite. Le informazioni sulle prestazioni della memoria sostenuta non sono tipicamente rese disponibili dai produttori per cui necessitano di essere misurate direttamente. A questo scopo è stato creato il benchmark STREAM, ed usa un maggior numero di dati di quelli che possano entrare nella cache in questo modo i risultati dovrebbero essere più indicativi di applicazioni dedicate principalmente al calcolo vettoriale, e si è obbligati a prelevare dati dalla memoria centrale e non da memorie ad un livello inferiore di gerarchia evitando di sfruttare meccanismi di caching e quindi falsare i risultati. Il nucleo di STREAM si occupa di mandare in esecuzione quattro kernel che in maniera diversa sfruttano il bandwidth: *COPY* effettua una copia di vettori, *SCALE* che prima di copiare una valore di un vettore nella sua posizione corrispondente lo moltiplica per un valore costante, *SUM* che effettua la somma di due vettori, *TRIAD*, effettua una somma ed una moltiplicazione dando in output un nuovo vettore a, da notare infatti che l'operazione non è paragonabile alla nota *SAXPY* in quanto viene utilizzato un nuovo vettore. Questi nuclei intendono rappresentare le operazioni elementari su cui si basano i programmi che fanno uso di vettori e matrici, e sono utilizzati in maniera specifica per diminuire al massimo il riuso dei dati [26]. In riferimento al modello espresso in 1.4, la prestazione sostenuta del bandwidth di memoria in STREAM viene calcolata come in

1.9. Ovvero come:

$$B_{actual} = \frac{N_{mem}}{T_{sw}}$$

Listing 2.1: I nuclei computazionali utilizzati da STREAM

```
-------------------------------------------------------------
    name         kernel              bytes/iter      FLOPS/iter
-------------------------------------------------------------
    COPY:     a(i) = b(i)                16               0
    SCALE:    a(i) = q*b(i)              16               1
    SUM:      a(i) = b(i) + c(i)         24               1
    TRIAD:    a(i) = b(i) + q*c(i)       24               2
-------------------------------------------------------------
```

2.2.2 Risultati di STREAM [ARK1]

Di seguito vengono illustrati i risultati ottenuti sul Mont-Blanc di STREAM. Per mandare in esecuzione in maniera sensata il benchmark stream è necessario ottenere il dato inerente alla cache L3 del processore in oggetto poiché l'intento è quello di instanziare un numero di elementi superiore a quello che la cache L3 sia in grado di contenere. Nel caso del Mont-Blanc, i cui processori sono degli ARM Cortex-A15, la cache L3 non è presente per questo motivo si considera la cache L2 come cache di livello più alto disponibile che ha una capienza pari a 4MB per questo motivo è necessario impostare la variabile *STREAM_ARRAY_SIZE* circa a *2000000*. Inoltre è possibile utilizzare STREAM sia su monoprocessore sia su multicore o distribuito (anche se quest'ultima è ancora in fase di sviluppo). Per la versione multicore è possibile osservare l'utilizzo della banda al variare dei thread in esecuzione. In tabella 2.6 è possibile osservare i risultati su monoprocessore.

E i risultati con dimensione fissata di 2000000 con due thread in tabella

	20 Milioni	*AVG Time*	*120 Milioni*	*AVG Time*
Copy	5945.4	0.053863	5600.1	0.343033
Scale	5889.2	0.054400	5600.1	0.344740
Add	4966.1	0.096767	4718.3	0.611107
Triad	4941.0	0.097252	4657.1	0.618673

Tabella 2.6: Risultati per STREAM

2.7, impostato in maniera tale da sfruttare l'intera potenza computazionale messa a disposizione dall'ARM A15 con due core.

Function	*Best Rate MB/s*	*Avg time*	**Min time**	**Max time**
Copy	6768.1	0.004788	0.004728	0.004823
Scale	6520.2	0.004929	0.004908	0.004951
Add	5713.7	0.008450	0.008401	0.008522
Triad:,	5705.4	0.008492	0.008413	0.008579

Tabella 2.7: Risultati di STEAM usando i due core del CortexA15

Capitolo 3

Strumenti per valutare l'applicazione

In questo capitolo si illustrano gli strumenti usati per la valutazione delle prestazioni di applicazioni. (Figura 3.1)

3.1 Valutazione delle prestazioni

Cosa è necessario misurare e perché?

Una macchina $M1$ ha prestazioni migliori di una macchina $M2$ se:

$$T^1_{sw}(A) \leq T^2_{sw}(A)$$

dove $T^1_{sw}(A)$ è il tempo del software che implementa l'algoritmo A sulla macchina $M1$ e $T^2_{sw}(A)$ è il tempo dello stesso software sulla macchina $M2$. Dalla (1.4) le quantità che è necessario misurare sono:

- $T_{sw}(A)$, il tempo di esecuzione dell'algoritmo;
- N_{flop}, il numero di operazioni del software;
- N_{mem}, il numero di accessi alla memoria.

Le tecniche che si utilizzeranno sono:

- Interval timing, la misurazione dei tempi del software mediante instrumentazione per ottenere $T_{sw}(A)$;

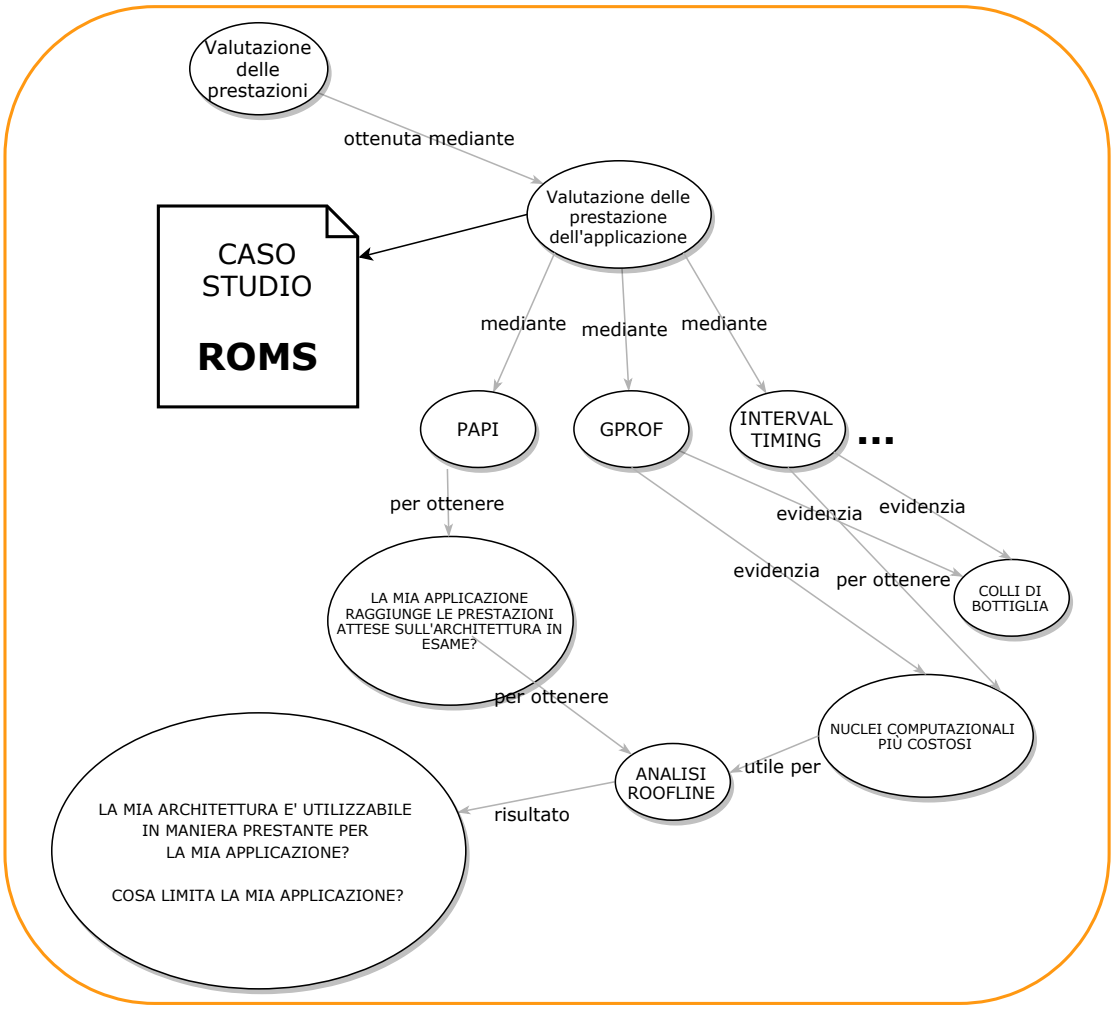


Figura 3.1: In questo capitolo si illustrano alcuni degli strumenti utili alla compresione della valutazione delle prestazioni di una applicazione

- Conteggio degli eventi, che colleziona dati generati dai contatori hardware [1] per ottenere N_{flop} ed N_{mem}.

Chiaramente, in riferimento alla tecniche del conteggio degli eventi, è possibile applicare tale tecniche all'intero programma ma ha senso concentrarsi solo sui nuclei computazionali che hanno un peso maggiore all'interno dell'esecuzione, perché migliorando le sezioni più costose si migliora di conseguenza l'intero software. Per questo motivo si parlerà anche della tecnica del *profiling* del codice sorgente. Il Profiling (o sampling) consiste nella raccolta dinamica di informazioni sul comportamento del software per capire quali sono le parti del software che dominano l'esecuzione.

Interval timing

Interval Timing (o anche detto intervallo temporale) combina le misure dei contatori con il tempo trascorso. A differenza delle tecniche che utilizzano la campionatura del program counter, l'interval timing calcola l'ammontare del tempo speso in frammenti di codice, con una coppia di chiamate alle funzioni che calcolano il tempo, una posta all'inizio della parte del sorgente da monitorare e l'altra posta alla fine.
La prima invocazione registra il tempo di ingresso; la seconda usa il tempo corrente e il tempo precedente per determinare il tempo trascorso. I dati sul tempo possono essere usati per calcolare il tempo totale di esecuzione di frammenti di codice, ciò può essere fatto sommando gli intervalli temporale in determinati blocchi del sorgente. Alternativamente si può semplicemente registrare la grandezza di ciascun intervallo e post-processare i dati ottenuti. Comunque, per sistemi massivamente paralleli, registrare ciascun intervallo su ciascun task può facilmente produrre un grande quantitativo di dati. quindi, l'interval timing è spesso usato su grandi sezioni di codice che dominano l'esecuzione oppure si usa per sommare il tempo speso in frammenti di codice invocati frequentemente ma di piccole dimensioni.

Conteggio degli eventi

Tipicamente, l'event counting (o anche conteggio degli eventi) è usato per calcolare il numero di volte che vengono eseguiti una procedura o altri frammenti del codice sorgente. In sostanza il compilatore può inserire contatori

[1] I contatori hardware anche chiamati *hardware performance counters*, sono un insieme di registri speciali contenuti nei moderni processori che conservano le informazioni sull'attività che dell'harwdare nel sistemi di calcolo.

in parti di codice per generare dati relativi all'esecuzione.
Per effettuare catture non-intrusive dei dati relativi alle prestazioni dell'hardware, molti produttori di microprocessori includono contatori disponibili su chip, quindi direttamente disponibili nell'hardware. Questi contatori possono essere letti e resettati mediante controllo software, ma vengono incrementati automaticamente quando avviene una certa operazione hardware. Ciascun contatore viene automaticamente aggiornato dal processore, quindi non c'è alcuna overhead per la registrazione. Mentre l'aggiornamento dei contatori via software richiede anche monitoraggio passivo o instrumentazione invasiva (per esempio mediante instrumentazione software di programmi per contare l'esecuzione i frammenti di codice). Gli strumenti allo stato dell'arte utilizzano delle librerie mediante API (Application Programming Interface) per rendere semplice l'accesso ai contatori. La cattura degli eventi è utile per ottenere il numero di operazioni floating point eseguite durante l'esecuzione od il numero di dati prelevati dalla memoria principale. PAPI è la libreria maggiormente utilizzato a questo scopo.

Profiling

Un metodo comune di monitoraggio è il profiling (o anche chiamata sampling) del program counter. Il sampling è ampiamente usato tramite strumenti come prof e gprof e mostra il comportamento relativo a ciascuna routine del software raffigurato con un istogramma che viene aggiornato periodicamente effettuando la campionatura del program counter durante l'esecuzione del software. L'inizializzazione del programma crea automaticamente un buffer (un'area di memoria temporanea per memorizzare le informazioni sulla campionatura) per conservare l'istogramma dei dati. Ciascuna riga dell'istogramma corrisponde ad una suddivisione di egual grandezza delle varie routine di un software. Durante l'esecuzione del programma, l'interrupt, ovvero la fase in cui viene campionato il program counter, viene effettuato ad intervalli regolari, in istanti di tempo prefissati. In ciascuna interruzione, il sistema campiona il program counter e incrementa la parte dell'istogramma appropriata. Quando il programma termina l'esecuzione l'istogramma dei dati viene salvato su un file. Siccome la campionatura avviene a intervalli regolari, la riga dell'istogramma può essere usata per calcolare il quantitativo di tempo speso in ciascuna procedura. Per ottenere un profilo accurato, il tempo totale di esecuzione deve essere sufficientemente alto per accumulare un insieme di esempi statisticamente utili, poiché alcune parti eseguite rapidamente potrebbero non essere quantificabili.

3.2 PAPI: Performance Application Programming Interface

PAPI è un acronimo per '**P**erformance **A**pplication **P**rogramming **I**nterface'. L'API è stata sviluppata dall'Innovative Computing Laboratory dell'università del Tennessee. L'idea è quella di accedere ai contatori hardware già disponibili nei moderni microprocessori.
I contatori hardware sfruttati da PAPI sono presenti nella maggior parte dei processori moderni. Possono fornire una base da cui partire per misurare le parti del software interessanti per lo studio delle prestazioni. Prima di PAPI esistevano solo poche API in grado di accedere a tali contatori ed erano scarsamente documentati ed instabili. Lo svantaggio offerto è che si potrebbero avere diverse metriche e definizione di contatori in base ai processori utilizzati [27],[28], mentre PAPI fornisce un accesso univoco ai contatori hardware rendendone standard l'utilizzo. L'insieme di queste considerazioni motiva lo studio di PAPI:

- Fornire una base solida e multipiattaforma per l'analisi delle performance
- Creare un insieme di definizioni standard per le metriche relative alle performance
- Fornire una API standard per utenti, produttori e accademici.
- Facile da usare, ben documentata, e liberamente utilizzabile.

PAPI fornisce due interfacce relative a ciascun contatore hardware (3.3):

- High Level API: tale interfaccia è di alto livello e fornisce tutti gli strumenti utili per accedere alle informazioni più semplici relative ai contatori. Essa fornisce la capacità di iniziare, fermare e leggere i contatori.
- Low Level API: Le API di basso livello gestiscono la lettura dei contatori hardware in insiemi definiti dall'utente chiamati EventSet.

il motivo è quello di fornire una buona portabilità tra le varie architetture dove possibile. Ovvero l'insieme delle API di alto livello costituiscono già una buona base da cui partire per costruire le proprie misure.Se si decide invece di effettuare analisi più profonde e specifiche si ha la possibilità di scendere nel dettaglio con le API di basso livello. In questo paragrafo si intende illustrare il funzionamento della libreria PAPI per capirne l'utilizzo,

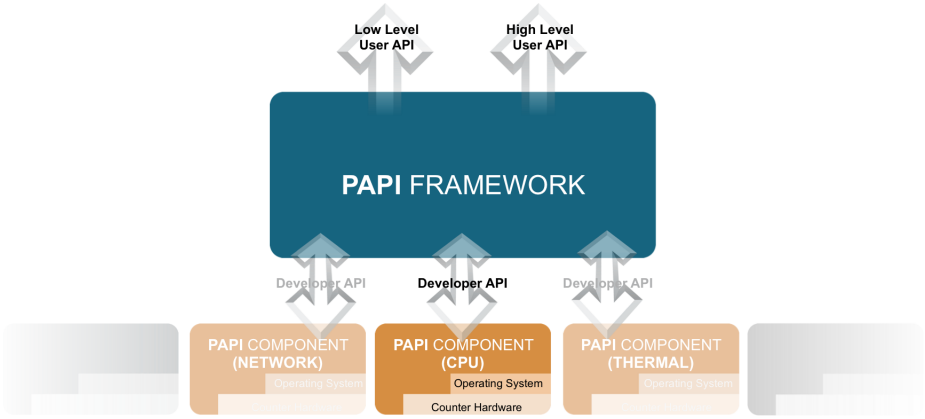


Figura 3.2: Architettura software della libreria PAPI

nell'esempio 3.1 si mostra l'utilizzo delle routine di alto livello (High Level API) in cui viene effettuato un prodotto matrice matrice.

Listing 3.1: Un esempio base per ottenere i FLOPS con le routine PAPI di alto livello

```
/*
 * Un semplice esempio di utilizzo delle routine PAPI_flops
     di alto livello.
  * PAPI_flops misura il tempo trascorso, il tempo del
     processo, le istruzioni floating-point
  * e i MFLOP/s.
 */

        ...
#include "papi.h"

#define INDEX 100

        ...

int main(int argc, char **argv) {
  float matrixa[INDEX][INDEX], matrixb[INDEX][INDEX],
     mresult[INDEX][INDEX];
  float real_time, proc_time, mflops;
  long long flpins;
  int retval;
  int i,j,k;

  /* Inizializzazione delle matrici */
```

```
  for ( i=0; i<INDEX*INDEX; i++ ){
    mresult[0][i] = 0.0;
    matrixa[0][i] = matrixb[0][i] = rand()*(float)1.1; }

  /* Setup della libreria PAPI e collezione dei dati */
  if((retval=PAPI_flops( &real_time, &proc_time, &flpins, &
      mflops))<PAPI_OK)
    test_fail(__FILE__, __LINE__, "PAPI_flops", retval);

  /* Prodotto matrice matrice */
  for (i=0;i<INDEX;i++)
   for(j=0;j<INDEX;j++)
    for(k=0;k<INDEX;k++)
      mresult[i][j]=mresult[i][j] + matrixa[i][k]*matrixb[k
          ][j];

  /* Inserire i dati dei contatori nelle variabili */
  if((retval=PAPI_flops( &real_time, &proc_time, &flpins, &
      mflops))<PAPI_OK)
    test_fail(__FILE__, __LINE__, "PAPI_flops", retval);

  printf("Real_time:\t%f\nProc_time:\t%f\nTotal flopins:\t%
      lld\nMFLOPS:\t\t%f\n",
  real_time, proc_time, flpins, mflops);
  printf("%s\tPASSED\n", __FILE__);
  PAPI_shutdown();
  exit(0);
}

        ...
}
```

Per compilare l'eseguibile è sufficiente aggiungere il riferimento alla libreria -lpapi, se correttamente installata nel proprio sistema:

```
gcc -o PAPI_exe PAPI_source.c -lpapi
```

L'output ottenuto per questo tipo di esempio:

```
Real_time:      0.009535
Proc_time:      0.009535
Total flopins:  2036215
MFLOPS:         213.551651
PAPI_flops.c    PASSED
```

come è possibile osservare, l'introduzione di PAPI mediante API di alto livello è molto semplice è sufficiente utilizzare le routine:

- PAPI_flops: che si occupa di inizializzare la libreria e raccogliere dati dai contatori hardware. Inserendo le chiamate a questa routine prima e dopo il blocco utile alla valutazione.

L'output mostra il tempo 'reale' occupato dal processo e il tempo effettivamente impiegato dal processo in 'proc_time'. Il valore su cui focalizzare l'attenzione è il numero di MFLOPS ottenuti ovvero la prestazione sostenuta (definita in 1.5).
PAPI è in grado di collezionare gli eventi che accadono su una cpu o su altri sottosistemi. Ci sono normalmente più eventi che potrebbero essere misurati rispetto ai contatori, quindi le API forniscono anche i mezzi per mappare gli eventi nei contatori.
In più gli eventi sono nativi di ciascun componente, PAPI definisce un insieme di eventi standardizzati su tutte le componenti CPU. per facilitare la scoperta di nuovi eventi PAPI fornisce un modo per interrogare il sistema circa la disponibilità di determinati eventi (papi_avail). Al nostro scopo, ovvero quello di valutare alcuni aspetti delle prestazioni di software paralleli, risulta necessario l'utilizzo degli eventi dato che le API di alto livello non sono thread safe, dato lo scopo per cui vengono utilizzati gli eventi i risultati potrebbero risultare falsati in quanto c'è la necessità di ottenere le operazioni di tutti i processi coinvolti nell'esecuzione nel caso in cui PAPI venga utilizzato per software paralleli.
Esiste anche la possibilità di utilizzare PAPI per analizzare software parallelo scritto con la libreria MPI. Di seguito si mostra un utilizzo base di PAPI:

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

int main(argc,argv)
int argc;
char *argv[];
{
  int done = 0, n, myid, numprocs, i, rc, retval, EventSet =
      PAPI_NULL;
  double PI25DT = 3.14159265358979323846264 3;
  double mypi, pi, h, sum, x, a;
  long_long values[1] = {(long_long) 0};

  MPI_Init(&argc,&argv);
  MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
  MPI_Comm_rank(MPI_COMM_WORLD,&myid);

  /*Inizializzazione libreria */
  retval = PAPI_library_init(PAPI_VER_CURRENT);
  if (retval != PAPI_VER_CURRENT) handle_error(retval);
```

```
/* Creazione dell'EventSet */
retval = PAPI_create_eventset(&EventSet);
if (retval != PAPI_OK) handle_error(retval);

/* Aggiungere l'evento sulle istruzioni all'EventSet */
retval = PAPI_add_event(EventSet, PAPI_TOT_INS);
if (retval != PAPI_OK) handle_error(retval);

/* Inizio conteggio */
retval = PAPI_start(EventSet);
if (retval != PAPI_OK) handle_error(retval);

while (!done)
{
  if (myid == 0) {
      printf("Inserisci un numero: (0 quits) ");
      scanf("%d",&n);
  }
  MPI_Bcast(&n, 1, MPI_INT, 0, MPI_COMM_WORLD);
  if (n == 0) break;

  h   = 1.0 / (double) n;
  sum = 0.0;
  for (i = myid + 1; i <= n; i += numprocs) {
      x = h * ((double)i - 0.5);
      sum += 4.0 / (1.0 + x*x);
  }
  mypi = h * sum;

  MPI_Reduce(&mypi, &pi, 1, MPI_DOUBLE, MPI_SUM, 0,
      MPI_COMM_WORLD);

  if (myid == 0)
      printf("pi e' approssimativamente %.16f, Error is
          %.16f\n",
              pi, fabs(pi - PI25DT));
  }

 /* Leggi i contatori */
 retval = PAPI_read(EventSet, values) != PAPI_OK);
 if (retval != PAPI_OK) handle_error(retval);

 printf("Dopo aver letto i contatori: %lld\n",values[0]);

 /* Stop dei contatori */
 retval = PAPI_stop(EventSet, values);
 if (retval != PAPI_OK) handle_error(retval);
 printf("Dopo aver fermato i contatori: %lld\n",values[0])
     ;
```

```
    MPI_Finalize();
}
```

questo esempio esegue il calcolo del π in parallelo tra più unità processanti, la libreria PAPI in questo caso è in grado di collezionare i dati dei contatori di ciascun processore coinvolto nell'esecuzione.
In tabella 3.1 sono ripotate alcune delle metriche basilari filtrate in base a quelle che possono avere una certa rilevanza. Per un elenco completo è possibile riferirsi a [29]. Mediante l'utilizzo delle metriche messe a disposizione

Name	*Description*
Floating Point Operations:	
PAPI_FMA_INS	FMA instructions completed
PAPI_FML_INS	Floating point multiply instructions
PAPI_FNV_INS	Floating point inverse instructions
PAPI_FP_INS	Floating point instructions
PAPI_FP_OPS	Floating point operations
PAPI_FP_STAL	Cycles the FP unit
Instruction Counting:	
PAPI_FXU_IDL	Cycles integer units are idle
PAPI_INT_INS	Integer instructions
PAPI_TOT_CYC	Total cycles
PAPI_TOT_IIS	Instructions issued
PAPI_TOT_INS	Instructions completed
PAPI_VEC_INS	Vector/SIMD instructions
Cache Access:	
PAPI_L1_DCA	L1 data cache accesses
PAPI_L1_DCH	L1 data cache hits
PAPI_L1_DCM	L1 data cache misses
PAPI_TLB_DM	Data translation lookaside buffer misses
PAPI_TLB_IM	Instruction translation lookaside buffer misses
PAPI_TLB_SD	Translation lookaside buffer shootdowns
PAPI_TLB_TL	Total translation lookaside buffer misses

Tabella 3.1: PAPI metrics: Metriche base accessibili mediante PAPI

da PAPI si possono ottenere le cosiddette metriche derivate, alcuni esempi in 3.2, ovvero eventi non direttamente disponibili nelle metriche base ma ottenibili mediante l'intersezione di queste.

Metric	*Formula*
Instructions	
Percentage floating point instructions	PAPI_FP_INS/PAPI_TOT_INS
L1 instruction cache miss ratio	PAPI_L2_ICR/PAPI_L1_ICR
Cache & Memory Hierarchy	
L1 cache data hit rate	1.0 - (PAPI_L1_DCM/PAPI_LST_INS)
L1 data cache read miss ratio	PAPI_L1_DCM/PAPI_L1_DCA
Bandwidth used (Lx cache)	((PAPI_Lx_TCM * Lx_linesize) / PAPI_TOT_CYC) * Clock(MHz)
Aggregate Performance	
MFLOPS (CPU cycles)	(PAPI_FP_INS/PAPI_TOT_CYC) * Clock(MHz)
MFLOPS (effective)	PAPI_FP_INS/Wallclock time
MIPS (CPU cycles)	(PAPI_TOT_INS/PAPI_TOT_CYC) * Clock(MHz)
MIPS (effective)	PAPI_TOT_INS/Wallclock time
Processor utilization	(PAPI_TOT_CYC*Clock) / Wallclock time

Tabella 3.2: PAPI metrics: Alcune metriche derivate ottenibili dalle metriche base fornite da PAPI

3.3 GPROF

GPROF è uno strumento di analisi delle prestazioni per piattaforme UNIX, utilizza l'istrumentazione per ottenere informazioni sul codice sorgente che interessa ottimizzare. Serve a capire quali parti del software occupano la maggior parte dell'esecuzione mediante un output che mostra il tempo totale occupato da ciascuna routine all'interno del sorgente e il numero di volte che essa viene richiamata (3.2), un profiler come GPROF è in grado di fornirci in maniera leggibile i possibili colli di bottiglia presenti nel software. GPROF è in grado di inserire codice automaticamente all'interno del software che si intende analizzare [2]. E' sufficiente inserire in fase di compilazione:

```
$gfortran -pg source.F -o exe
```

successivamente si esegue il software normalmente

```
$./exe
```

generando il file traccia binario gmon.out, utilizzando il comando gprof si può leggere il contenuto del file traccia dando in input l'eseguibile ed il file traccia stesso generato durante l'esecuzione:

```
$gprof ./exe ./gmon.out > output.txt
```

Nell'esempio fornito si ottiene un file di testo di questo tipo:

Listing 3.2: Output di GPROF su un programma d'esempio

```
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  ms/call  ms/call  name
 33.34      0.02     0.02     7208     0.00     0.00  open
 16.67      0.03     0.01      244     0.04     0.12
    offtime
 16.67      0.04     0.01        8     1.25     1.25
    memccpy
 16.67      0.05     0.01        7     1.43     1.43  write
 16.67      0.06     0.01                             mcount
  0.00      0.06     0.00      236     0.00     0.00  tzset
  0.00      0.06     0.00      192     0.00     0.00
     tolower
  0.00      0.06     0.00       47     0.00     0.00  strlen
  0.00      0.06     0.00       45     0.00     0.00  strchr
  0.00      0.06     0.00        1     0.00    50.00  main
  0.00      0.06     0.00        1     0.00     0.00  memcpy
  0.00      0.06     0.00        1     0.00    10.11  print
  0.00      0.06     0.00        1     0.00     0.00  profil
  0.00      0.06     0.00        1     0.00    50.00  report
```

open è la funzione che impiega più tempo nell'esecuzione totale (33%) il cui peso è dato dal tempo speso nella singola chiamata moltiplicato per il numero di volte che viene chiamata. Le funzioni sono ordinate secondo il parametro *self seconds* ovvero il numero di secondi speso da ciascuna funzione senza considerare tutte le altre chiamate anche interne ad essa.

3.4 Un modello per le prestazioni: Roofline

Roofline è un modello di prestazioni visuale (sviluppato dal Berkeley Lab [30]) e si pone come obiettivo quello di fornire un modello grafico, quindi di maggiore facilità di lettura rispetto a dati in forma tabella, che fornisce e riassume in forma direttamente comprensibile le prestazioni di un software. Cosa fornisce Roofline?

- Un parametro tenuto in considerazione dal modello è l'intensità delle comunicazioni, ciò che viene valutato qui è la **larghezza di banda della memoria** (definita in 1.17, par.1 - pag.13), misurata in Gigabyte per secondo (miliardi di byte trasferiti al secondo). Il motivo per cui viene valutato è questo aspetto è dovuto al fatto che la memoria ha una grande incidenza sul tempo di risoluzione di un algoritmo poiché le unità aritmetiche devono operare su dati scritti in memoria. Per

cui si ricorre a meccanismi di caching per rendere la fase di lettura e scrittura dei dati da e per la memoria più rapidi. Inoltre anche le ottimizzazioni dal punto di vista software possono aiutare ad evitare dei cache miss quindi a minimizzare le comunicazioni inutili da e verso le memorie.

- Altro aspetto è la **locazione della memoria**: in caso di memoria distribuite più esse saranno numerose maggiore è la probabilità di un aumento delle comunicazioni; per questo è necessario applicare le tecniche per favorire la località dei dati.

Il modello Roofline consiste in un grafico/tabella che considera le caratteristiche descritte precedentemente, e il fulcro di questo modello è quello di mettere in relazione i GFLOPs con i GByte/s con una quantità chiamata intensità aritmetica (in letteratura anche definita col nome di *Computational Intensity* [31]). Calcolata come espresso in 3.5. L'intensità aritmetica è la stima del numero di richieste che un software effettua alla memoria, o meglio è il numero di operazioni in virgola mobile per byte di memoria a cui è richiesto l'accesso alla memoria. In Figura 1.5 si possono osservare i diversi valori di intensità aritmetica (definita in 3.5, par. 1 - pag.14) che gli algoritmi base solitamente utilizzati nel calcolo scientifico hanno, ad esempio operazioni relative a livello 1 delle librerie BLAS (BLAS1), ad esempio un incremento vettore-vettore ($x(i) += y(i)$) ha una bassa intensità aritmetica circa 0.0417, ovvero N FLOPs rispetto a $24/N$ Bytes ed è indipendente dalla dimensione del vettore, ciò vuol dire che il c'è uno sbilanciamento delle operazioni effettuate rispetto a dati richiesti dalla memoria. Mentre invece nella parte destra della figura si possono osservare un'operazione con una intensità aritmetica più alta come ad esempio le routine incluse nel livello 3 delle BLAS (BLAS3) la cui intensità artimetica cresce più velocemente. Come si può osservare in Figura 3.4, a seconda del valore assunto dall'indice di intensità aritmetica, un kernel può avere un bound di prestazioni più o meno elevato. Si supponga ipoteticamente che il kernel analizzato sia il kernel 1 (Figura 3.4) esso sarà limitato superiormente dal bandwidth della memoria, ciò significa che effettua un numero maggiore di accessi alla memoria rispetto al numero totale di operazioni floating-point eseguite, invece il kernel 2 avendo una intensità aritmetica maggiore, non è limitato superiormente dal bandwidth di memoria, ed effettua un numero maggiore di operazioni. Più formalmente se un kernel è *memory-bound*, ovvero quando il tempo affinché ci sia un output dal software è pregiudicato principalmente dalla velocità e

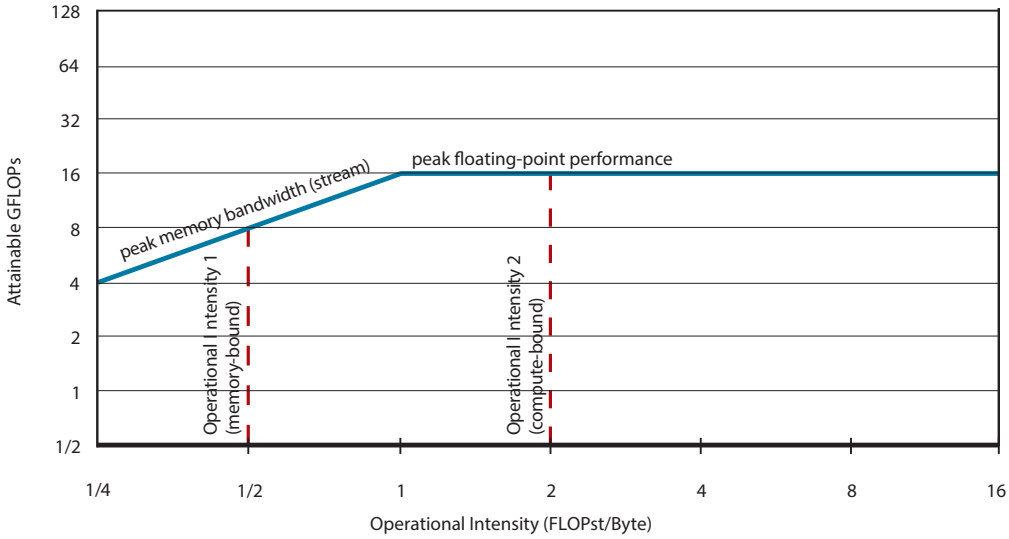


Figura 3.3: Grafico base prodotto mediante il modello Roofline [rielaborato a partire da [9]]

dalla capacità della memoria, si ha:

$$AI \leq \frac{P_{peak}}{B_{peak}} \tag{3.1}$$

dove P_{peak} è il valore che indica la performance di picco, mentre B_{peak} è il valore che indica il picco del badwidth di memoria. Dalla 3.1:

$$\frac{N_{flop}}{N_{mem}} \leq \frac{t_{mem}}{t_{flop}} \tag{3.2}$$

Se un kernel è *compute-bound* si ha che:

$$AI \geq \frac{P_{peak}}{B_{peak}} \tag{3.3}$$

quindi dalla 3.3:

$$\frac{N_{flop}}{N_{mem}} \geq \frac{t_{mem}}{t_{flop}} \tag{3.4}$$

Questa analisi ci può suggerire in quali direzioni sia utile dedicare lo sforzo per il miglioramento del kernel o del software tutto. Ad esempio il kernel 1 è collocato nella parte sinistra del grafico, oltre il punto di rottura anche detto *ridge point*, che indirizza a ridurre l'utilizzo della memoria, questo può

essere fatto aumentando l'intensità aritmetica, quindi sarebbe consigliabile una riscrittura (anche parziale) del codice. Il picco massimo è dato dalla seguente formula:

$$\frac{GFLOPs}{Sec} = min \begin{cases} AI \times B_{peak} \\ P_{peak} \end{cases} \tag{3.5}$$

una delle peculiarità del modello è anche questo, è in grado di suggerire ottimizzazioni che il programmatore può apportare al software per incrementare le prestazioni. Considerando l'esempio in Figura 3.4 le ottimizzazioni apportabili sono mostrate in Figura 3.5 e sono costruite in base all'architettura di riferimento. Quindi il modello roofline potrebbe suggerire le seguenti ottimizzazioni:

1. Far sì che ci sia un numero equo di operazioni di addizione e moltiplicazione eseguite simultaneamente, in modo da sfruttare al massimo l'hardware a disposizione.

2. Modificare il codice favorendo il parallelismo a livello di istruzioni per sfruttare al massimo l'architettura sottostante.

3. Caricare in anticipo le istruzioni in memoria senza aspettare che il flusso dati sia pronto, risparmiando così sul tempo di estrazione delle istruzioni.

4. Incrementare l'affinità di memoria e cioè far in modo che i dati e i relativi thread si trovino entrambi nella memoria dello stesso chip, senza dover andare in quella di altri e provocando così tempi di latenza maggiori. In linea di massima favorire la località dei dati.

In Fig.3.5 sono evidenziate le quattro ottimizzazioni sopracitate; si noti che in certe zone del grafico esse possono essere singole o combinazioni, ovvero un punto all'interno della zona colorata può corrispondere a più di una ottimizzazione da attuare [32]. Il modello roofline al primo passo fissa due limiti superiori (rappresentati in base alle due rette di colore azzurro in figura 3.5) che rappresentano la prestazione floating-point di picco e prestazione di picco massima del bandwidth della memoria. L'obiettivo del modello è quello di ottenere la prestazione raggiungibile definita come:

$$A_p\text{GFLOPs/sec} = min(P_{actual}, B_{peak} \times AI) \tag{3.6}$$

dove A_p è la prestazione raggiungibile (attainable), P_{actual} è la prestazione sostenuta e B_{peak} è la prestazione del bandwidth della memoria.

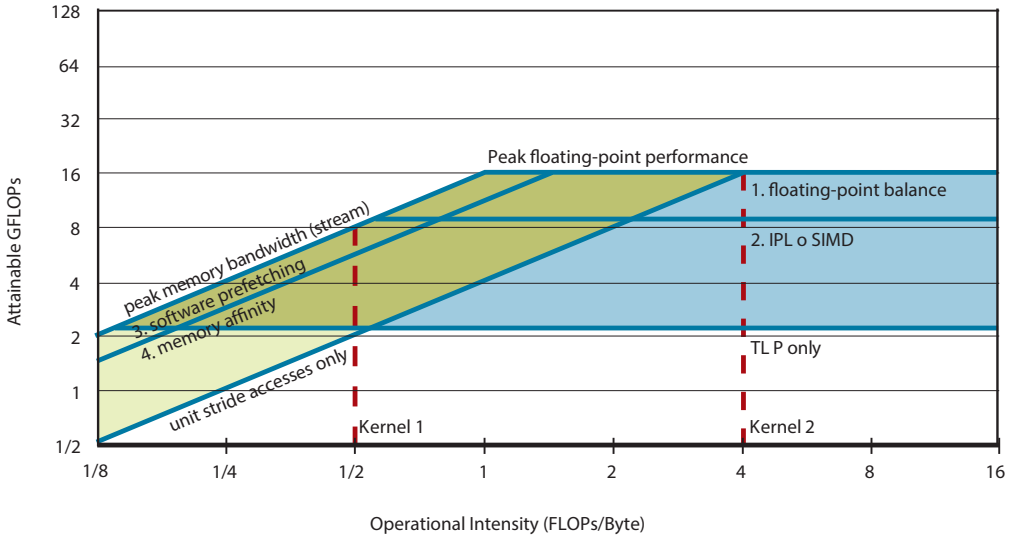


Figura 3.4: Ottimizzazione suggerite dal modello roofline [rielaborato a partire da [9]]

Quello mostrato precedentemente è un esempio di quali siano le possibili ottimizzazioni che un grafico Roofline può suggerire ma la costruzione dello stesso necessita di alcune delucidazioni.

Esempio 3.1 (Grafico Roofline su una architettura d'esempio) *Per rendere più chiara la costruzione del grafico può essere utile spiegarlo mediante un esempio. Avendo a disposizione l'architettura [ARK2] Intel® Core™ i5-3470S CPU @ 2.90GHz i dati sono osservabili in tabella 1.4 (sez. 1.2.2 - pag. 21). Il modello roofline può essere definito come:*

$$A_p = min(P_{actual}, AI * B_{peak}) \tag{3.7}$$

dove A_p rappresenta la prestazione attesa del nucleo computazionale analizzato. P_{actual} è la prestazione sostenuta dall'architettura target ottenuta considerandone le caratteristiche peculiari dell'architettura, AI è l'intensità aritmetica, B_{peak} è il picco del bandwidth della memoria. I dati del modello necessari sono:

- *P_{peak} la prestazione di picco teorica dell'architettura target;*
- *P_{actual} la prestazione sostenuta in base alle sue caratteristiche peculiari;*

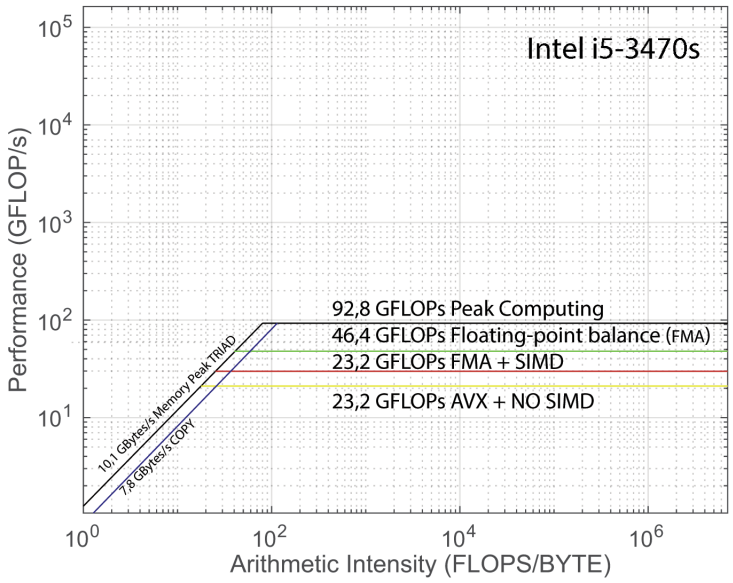


Figura 3.5: Grafico roofline per l'architettura Intel Sandy Bridge i5-3470s

- *AI intensità operazionale;*
- B_{peak} *badwidth di picco della memoria ottenuta mediante strumenti di micro-benchmarking.*

Come è possibile osservare in Figura 3.6 il picco computazionale rappresenta la prestazione di picco ottenuta in questo modo:

$$P_{peak} = 4\ core * 2,9\ GHz * 8\ op/cy = 92,8\ GFLOPs \tag{3.8}$$

essendo l'architettura dotata di 4 core che effettuano 8 operazioni floating-point per ciclo di clock, dati disponibile nel manuale dell'archiettura [13], il picco del bandwidth di memoria è ottenuto con strumenti di micro-benhmarking come STREAM. Una volta stabili i due tetti dell'architettura si può procedere a calcolare ulteriori limiti. Per capire quali siano i limiti computazionali è necessario riferisci al manuale dell'architettura del produttore. Considerando quanto riporta il manuale Intel per le architetture SandyBridge [33] il processore Intel fornisce il supporto all'FMA (Fused Multiply Add) che esegue una moltiplicazione ed una addizione per cui senza utilizzare tale supporto si ottiene un nuovo limite stabilito come segue:

$$P_{FMA} = 4\ core * 2,9\ GHz * 4\ op/cy = 46,4\ GFLOPs \tag{3.9}$$

un ulteriore limite può essere fissato sfruttando il supporto all'FMA utilizzando un solo processore per l'esecuzione quindi:

$$P_{FMA} = 1\ core * 2,9\ GHz * 8\ op/cy = 23,2\ GFLOPs \tag{3.10}$$

un ulteriore limite può essere fissato utilizzando un solo processore senza il supporto all'FMA per cui si ottiene un ulteriore limite:

$$P_{FMA} = 1\ core * 2,9\ GHz * 4\ op/cy = 11,6\ GFLOPs \tag{3.11}$$

per cui il grafico risultante è osservabile in Figura 3.6.

Capitolo 4

Applicazione su un caso reale: ROMS

4.1 Assimilazione dei dati

Ormai è da tempo diffusa la simulazione di fenomeni fisici ottenuti attraverso modelli matematici, partendo da uno stato fisico ad un certo punto t si cerca di prevedere uno stato al tempo $t + \Delta t$ (con $\Delta t > 0$). L'assimilazione è un processo che si occupa di analizzare dei dati provenienti da osservazioni ed integrarli all'interno dello stato del modello. L'assimilazione dei dati è un processo applicabile a molti campi dalla geoscienza. Il processo di analisi può essere utilizzato a vari livelli:

- come approssimazione dello stato vero di un sistema fisico ad un dato tempo;
- come diagnosi comprensiva ed autoconsistente di un sistema fisico;
- come un riferimento attraverso il quale fare una verifica della qualità della osservazione;
- come un dato di input utile per un'altra operazione, ad esempio lo stato iniziale di un modello di previsione.

Nella data assimilation si hanno: le previsioni ottenute effettuando calcoli basati su un modello, e le osservazioni che provengono da sensori distribuiti nel globo (nel nostro caso nell'oceano), queste due tipologie di informazioni vengono comparate in tempo reale e danno luogo all'analisi [34], [35] come osservabile in Figura 4.1.

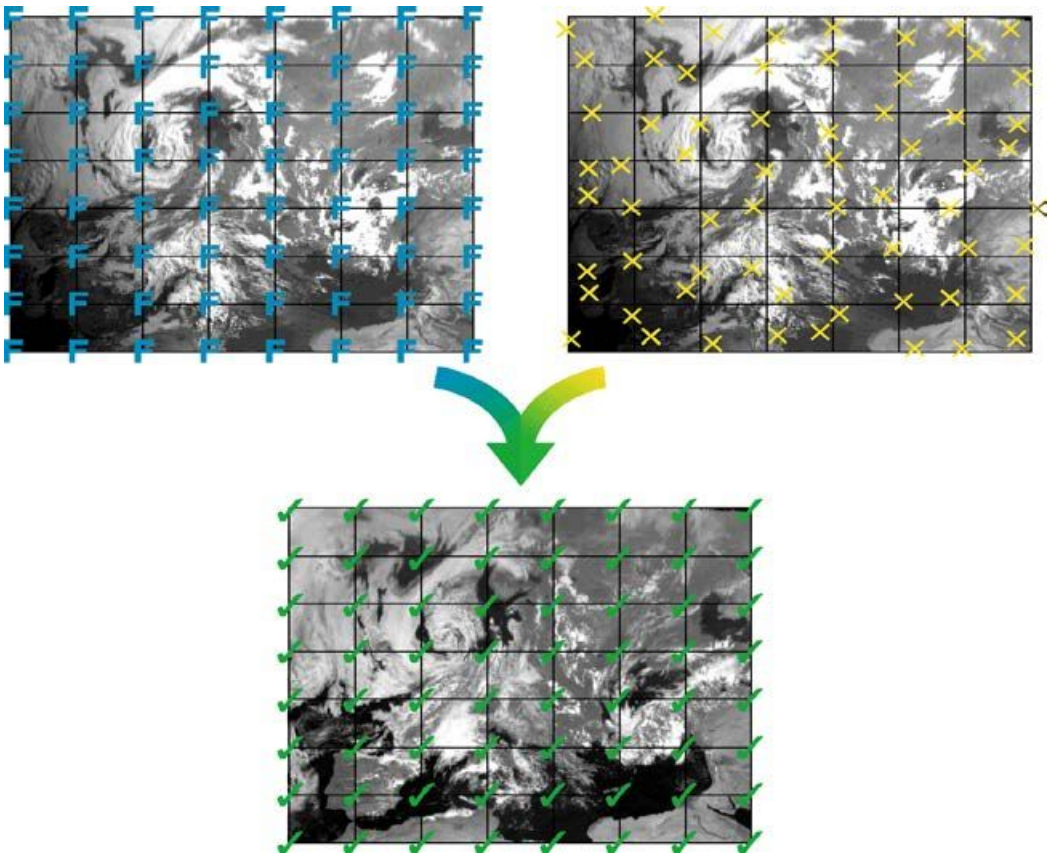


Figura 4.1: Il processo di assimilazione dei dati che genera l'analisi

Il modello fornisce consistenza ai dati osservati permettendo anche di interpolarli o estrapolarli in regioni dello spazio e del tempo in cui questi mancano. L'iter seguito dal processo di assimilazione è in grado di correggere la traiettoria prodotta durante il calcolo in iterazioni successive per questo i passi effettuati seguono questo iter: previsione-osservazione-correzione.

4.2 ROMS

Il modello ROMS (Regional Ocean Model System) è un modello ampiamente utilizzato dalla comunità scientifica per un insieme di applicazioni atte all'osservazione di: Correnti marine; Dinamica della criosfera; Trasporto dei sedimenti; Cicli tidali; Cicli biogeochimici Dispersione degli inquinanti. E' un modello open-source distribuito gratuitamente per studiare i modelli oceanici complessi [36]).
In Figura 4.2 è possibile osservare l'organizzazione per moduli del software. ROMS può essere eseguito sia in seriale che in parallelo. Il codice utilizza un paradigma di parallelizzazione a grana grossa (coarse-grained) che divide il dominio dei dati tridimensionali in tiles (partizioni). Ogni tile può essere mandata in esecuzione da un processo MPI, da un thread o, per la versione seriale, può essere eseguita da un solo processo in sequenziale. Ad ogni modo il software supporta sia l'esecuzione su shared memory che su distributed memory.

4.2.1 Organizzazione parallela del dominio: ROMS tiling

Come già accennato ROMS supporta esecuzioni seriali e parallele: sia shared con OpenMP, sia distributed con la libreria MPI (vedi Figura 4.3), l'utente ha la possibilità di scegliere tra le due opzioni durante la compilazione con opportune configurazioni. ROMS organizza il calcolo in **tile**, ovvero aree del dominio principale (Figura 4.3). Alcuni degli obiettivi della progettazione parallela di ROMS sono:

- Ridurre i tempi di esecuzione;
- Sfruttare l'hardware sottostante nella maniera migliore;
- Minimizzare i cambi del sorgente, infatti ROMS può essere utilizzato in parallelo o in distribuito con un FLAG in fase di avvio;
- Non inserire il numero dei processi a mano nel sorgente;
- Non cambiare l'implementazione seriale;

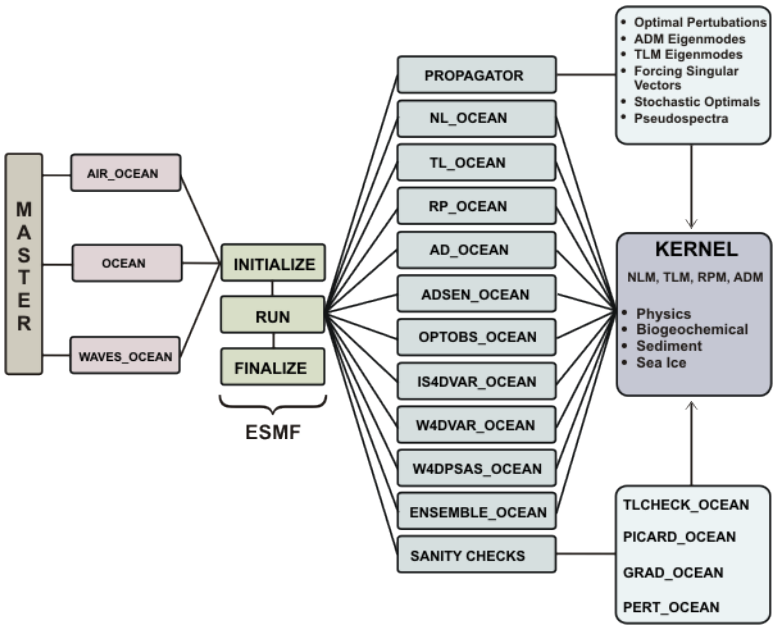


Figura 4.2: ROMS Framework [37]

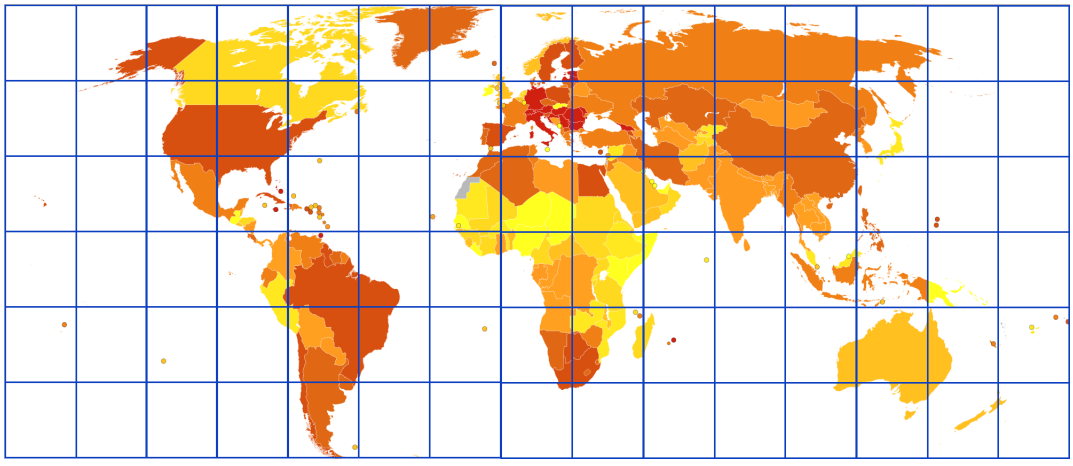


Figura 4.3: Tile parallele a partire dal dominio globale

- Stessi risultati del software seriale;
- Portabilità.

Ciascuna opzione viene gestita tramite il preprocessing. Ad esempio, compilando con OpenMP verrà aggiunta l'opzione -DMPI automaticamente. Ed all'interno del sorgente si troverà una definizione del tipo:

```
#if defined MPI
        #define DISTRIBUTE
#endif
```

che controlla l'attivazione o meno della parte parallelo del software. Ciò ha permesso di utilizzare rapidamente dei flag per le tre principali modalità d'esecuzione: *seriale, parallela mediante scambio di messaggi (MPI), parallela mediante thread (OpenMP)*. Una tipica suddivisione viene mostrata nella

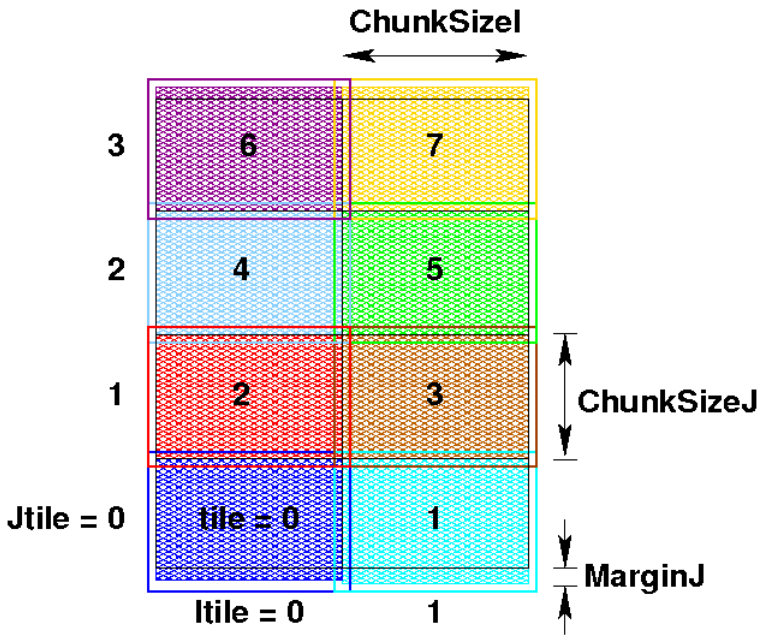


Figura 4.4: Tipica suddivisione dei tile di ROMS [Fonte: ROMS Wiki - Parallelization]

Figura 4.4 un colore per ciascun tile. Le aree che si sovrappongono vengono chiamate 'ghost points'. Ogni tile viene assegnata ad un processo, se ROMS è configurato per essere eseguito in un sistema a memoria condivisa allora ogni tile sarà assegnata ad un thread, mentre se è configurato per un sistema a memoria distribuita allora ciascun tile sarà assegnata ad un processo.

Le due configurazione non possono essere utilizzate contemporaneamente, quindi non si può utilizzare una modalità di esecuzione ibrida, ovvero sfruttando sia il parallelismo distribuito che quello a memoria condivisa.
Il numero di tile è configurabile nei file di input mediante le variabili **NtileI** ed **NtileJ**, se scegliessimo la configurazione MPI il prodotto $NtileI \times NtileJ$ dovrebbe essere uguale al numero dei processi MPI richiesti. Per esempio, per NtileI=4 e NtileJ=6, si devono avere un numero di processi MPI pari a 24. Ad ogni modo, anche usando il software seriale si possono avere 24 tiles che comunque verrano calcolate sequenzialmente. Una volta scelte queste due variaibili, la dimensione di ogni tile è calcolata in questo modo:

```
ChunkSizeI = (Lm+NtileI-1)/NtileI
ChunkSizeJ = (Mm+NtileJ-1)/NtileJ
MarginI = (NtileI*ChunkSizeI-Lm)/2
MarginJ = (NtileJ*ChunkSizeJ-Mm)/2
```

dove Lm ed Mm vengono usate solo nel caso di una esecuzione di ROMS a griglie innestate (nested grid), servono a gestire i punti interni a ciascun tile nelle due direzioni, le griglie innestate non sono oggetto di questa tesi quindi non sono state utilizzate e vengono trattate nella documentazione di ROMS [38].

4.3 Analisi delle prestazioni di ROMS

Ciascuno degli strumenti illustrati precedentemente è stato utilizzato per evidenziare i limiti e le caratteristiche migliorabili del software in particolare il modulo di ROMS che si occupa dell'assimilazione variazionale a 4 dimensioni.
I passi da seguire per la valutazione di un algoritmo parallelo sono i seguenti:

- Analisi delle prestazioni del sorgente del software originale, testando la scalabilità forte (definita in 1.30) su una architettura prefissata utilizzando la tecnica dell'interval timing illustrata nel paragrafo 3.2;
- Eseguire il profling del codice per identificare i colli di bottiglia del software illustrata nel paragrafo 3.2 con GPROF;
- Analizzare problemi di calcolo legati ai kernel (parti del sorgente) computazionalmente più onerosi utilizzando le tecniche illustrate in 3.2.

per intraprendere l'analisi si utilizzano tool specifici che facilitano il compito del calcolo della prestazione sostenuta, invece di effettuarlo manualmente come fatto nel 2.2.1. Andando a monitorare un solo nucleo computazionale si ha il vantaggio di non stravolgere il sorgente ma analizzare solo le parti che necessitano di una analisi più approfondita. Questo permette di ottenere miglioramenti con cambiamenti minori o comunque più efficaci ad ogni modo è comunque necessario instrumentare il software. Ad esempio, spesso potrebbe essere utile, specie nei software scientifico analizzare le librerie matematiche utilizzate in questo caso (ARPACK [39] LAPACK [17]) ottimizzando l'integrazione delle librerie a basso livello [40].

4.4 Monitoraggio delle prestazioni del software

Il primo passo da conseguire per monitorare le prestazioni di un software parallelo è osservarne l'esecuzione, in termini di tempo di risposta in modo da verificarne la scalabilità del software incrementando il numero di processori coinvolti nell'esecuzione [41].
L'algoritmo di ROMS relativo al problema dell'assimilazione variazionale incrementale a 4 dimensioni (IS4DVAR) consta dei passi osservabili in Figura 4.5. La prima fase si occupa di calcolare il modello non-lineare (NLROMS, legato alla variabile *outer-loop* che indica il numero di iterazioni che effettueranno nuovamente le stime di ciascun passo), successivamente viene calcolato il modello tangente (TLROMS, legato alla variabile *inner-loop* che indica il numero di stime che si ha la necessità di effettuare), infine viene calcolato il modello adjoint (ADROMS), l'ultima fase riguarda quella del calcolo di minimizzazione. Di base il software viene fornito con un esempio chiamato WC13 (Figura 4.6), configurato per la costa ovest della California e il CCS (California Current System) ovvero il suo sistema di correnti. Questa configurazione ha una risoluzione orizzontale pari a 30 km. L'ambiente su cui sono stati eseguiti i test è il cluster Mont-Blanc di Barcellona del Barcelona Supercomputing Center (BSC) (illustrati nelle tabelle 1.2 e 1.3), si è deciso di testare ROMS su una architettura distribuita per studiare il comportamento in ambito HPC. Si osservino in 4.1 i risultati dei test di scalabilità ottenuti variando il numero di processori coinvolti nell'esecuzione ed i grafici relativi nelle figure 4.7, 4.8 e 4.9.

Aumentando il numero di processori gradualmente è possibile ottenere una prima analisi sul comportamento del software, ma il test case WC13 non permette di aumentare il numero di processori oltre i 16 nodi per ragioni

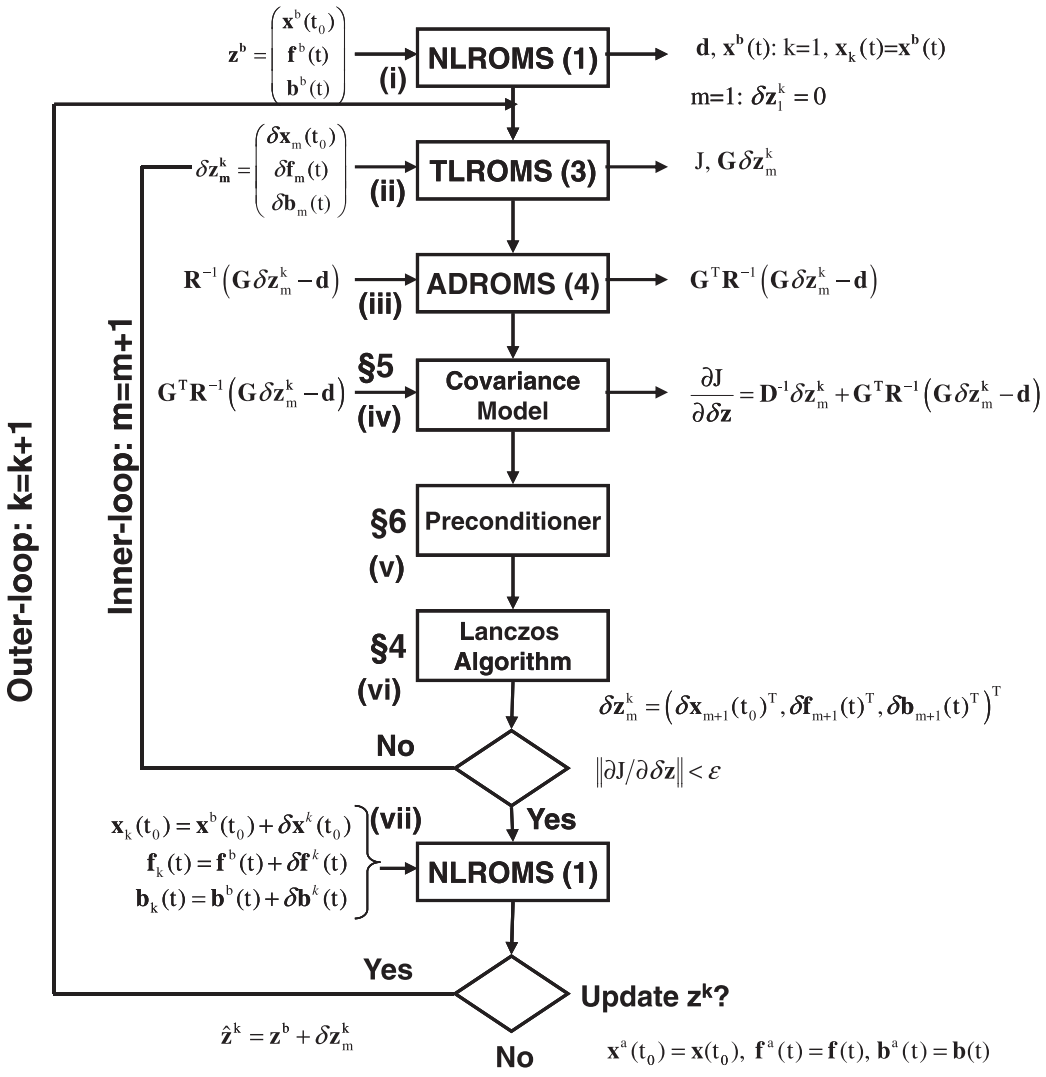


Figura 4.5: Passi presenti nel modulo IS4DVAR di ROMS [42]

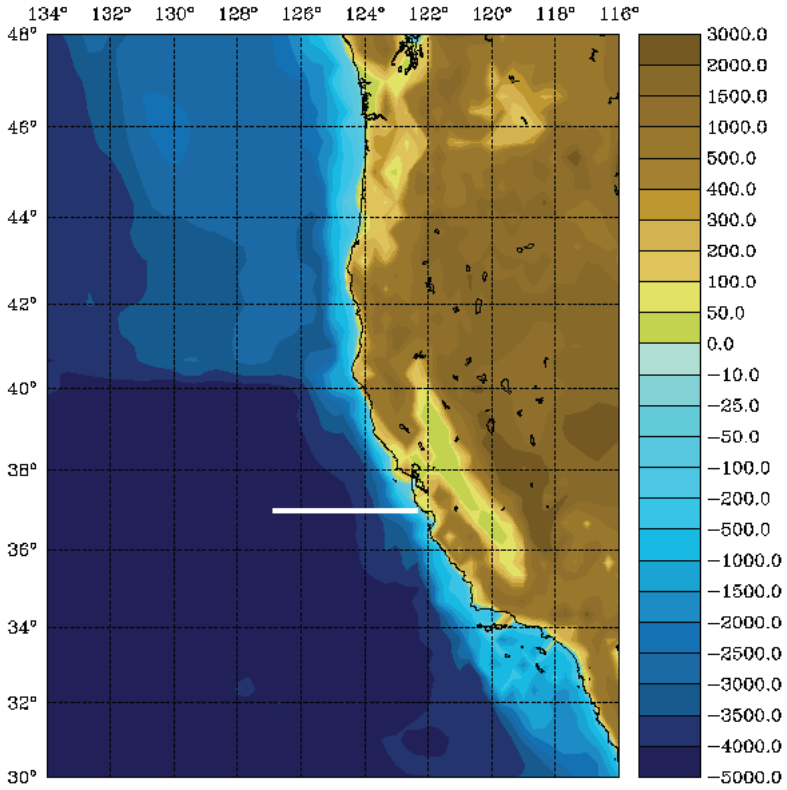


Figura 4.6: WC13 test case [Fonte: ROMS Tutorial - IS4DVAR]

	$T(sec)$	*Speed-up*	*Efficiency*
1	6839.5	1	1
2	4341.8	1.6	0.8
4	3137.4	2.1	0.6
8	2941.7	2.3	0.3
16	2246.5	3.0	0.2

Tabella 4.1: Risultati dell'esecuzione del software sul cluster Mont-Blanc al variare del numero dei processori

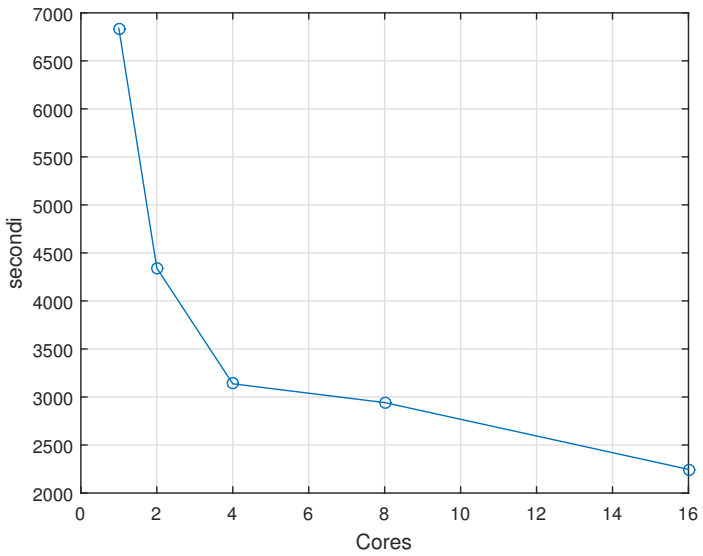


Figura 4.7: Scalabilità dell'algoritmo IS4DVAR di ROMS con il caso WC13

legate alla dimensione dell'input, è possibile quindi ricorrere ad un altro test case sviluppato dal gruppo di ricerca di oceanografia dell'UCSC derivante da WC13, chiamato WC12 riferito alle correnti della California la cui griglia ha una risoluzione maggiore per cui il numero di dati presenti è più alto, il test ha una precisione di 10km in direzione orizzontale. I test effettuati simulano 1 giorno e hanno una configurazione osservabile in tabella 4.2. Per maggiori informazioni sulla configurazione ci si riferisca all'appendice

Parametro	*Valore*	*Descrizione*
NTIMES	96	Numero totale di passi temporali dell'esecuzione
DT	900	dimensione dei passi temporali
$NTileI \times NTileJ$	$1, 2, 4, 6, 8, 10, 12$	Organizzazione del tiling
FLAGS	–freepack-arrays, -march=armv7-a -O3 –ffast-math	Flag utilizzati dal compilatore FORTRAN per l'ottimizzazione

Tabella 4.2: Configurazione utilizzata per il test WC12

B. I risultati ottenuti con il test case WC12 sono visionabili in tabella 4.7. Si nota, come nel test precedente, che il comportamento segue lo stesso

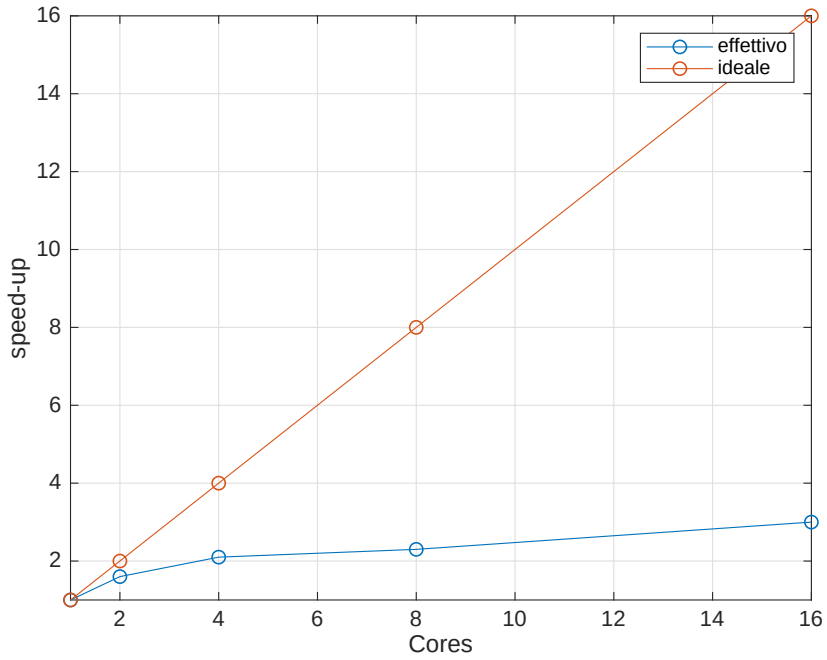


Figura 4.8: Speed up ottenuti con l'utilizzo di 1-2-4-8-16 e processori

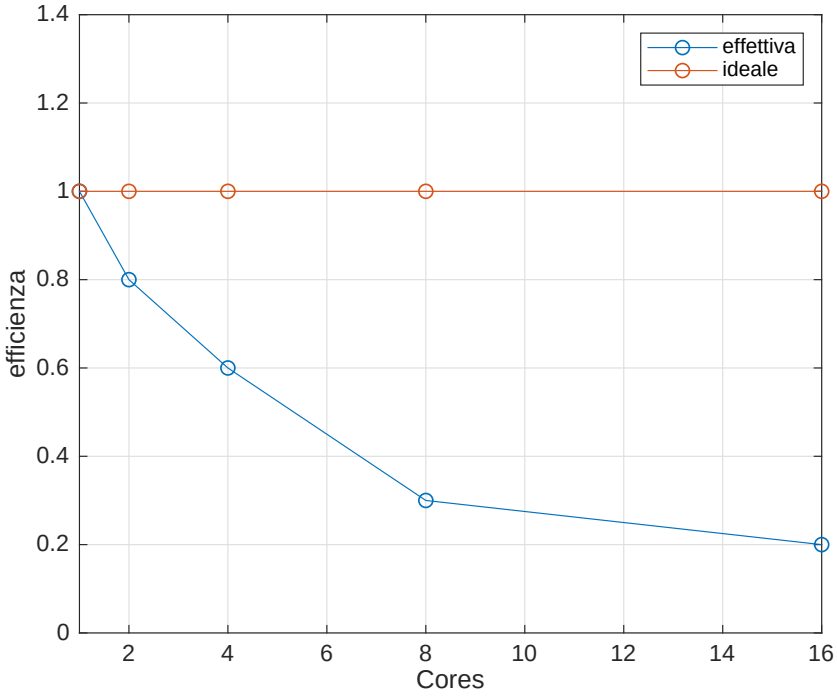


Figura 4.9: Efficienza per il caso WC13

Figura 4.10: ROMS performance su WC13

trend, osservando lo speed-up e di conseguenza l'efficienza tra 16 processori (dimensione massima utilizzabile con WC13) e 36 processori diminuisce dal 26% al 9%. Aumentando il numero di processori dai 36 processori ai 144 si nota come i tempi di esecuzione si stabilizzino sui 21000 secondi circa con un conseguente calo dell'efficienza. Ciò probabilmente accade poiché alcune delle latenze di memoria vengono nascoste e quindi i tempi per il calcolo si stabilizzano visto che la dimensione del problema resta fissata. Probabilmente se crescesse ulteriormente il numero di processori si noterebbe un peggioramento anche dello speed-up poiché i tempi per le comunicazioni coprirebbero una buona parte del tempo di esecuzione del software.
Il fatto che ROMS non scali bene si potrebbe imputare a problemi legati all'I/O per questo motivo sono state profilate le sezioni del software relative all'I/O (Tabella 4.3).

Frazione di sorgente	*Tempo*	*Percentuale*
NLROMS		
Allocation and array initialization	2.330	0.0023 %
Ocean state initialization	0.555	0.0005 %
Reading of input data	139.685	0.1381 %
Writing of output data	64.535	0.0638 %
TLROMS		
Ocean state initialization	4.475	0.0044 %
Reading of input data	1909.995	1.8884 %
Writing of output data	516.450	0.5106 %
ADROMS		
Ocean state initialization	4.275	0.0042 %
Reading of input data	1897.820	1.8764 %
Writing of output data	100.070	0.0989 %
Totale	4600	4.58 %

Tabella 4.3: Output di ROMS-WC12 relativo al tempo impiegato nello scambio di messaggi

Risulta chiaro come in questa configurazione il tempo speso nell'output da ROMS influisce molto poco nell'intera esecuzione, le sezioni che si occupano dell'I/O non superano il $4,58\%$ considerando tutti e tre i modelli del modulo *IS4DVAR* rispetto al peso delle altre parti.

Osservando il peso delle sezioni relative alle comunicazioni si nota come la percentuale di ROMS dedicata ad esse sia in totale di circa $8,43\%$, in particolare il modello **NLROMS** (4.4) pesa circa 0.43%, il modello **TL-**

Frazione di sorgente	*Tempo*	*Percentuale*
Profling scambio di messaggimodello Nonlinear (NLROMS)		
Message Passage: 2D halo exchanges	19.510	0.0193 %
Message Passage: 3D halo exchanges	23.840	0.0236 %
Message Passage: 4D halo exchanges	7.445	0.0074 %
Message Passage: lateral boundary exchanges	0.025	0.0000 %
Message Passage: data broadcast	345.120	0.3412 %
Message Passage: data reduction	3.545	0.0077 %
Message Passage: data gathering	7.800	0.3412 %
Message Passage: data scattering	28.530	0.0282 %
Message Passage: point data gathering	0.570	0.0006 %
Message Passage: synchronization barrier	0.005	0.0000 %
Totale	436.390	0.4315 %

Tabella 4.4: Output di ROMS-WC12 relativo al tempo impiegato nello scambio di messaggi

Frazione di sorgente	*Tempo*	*Percentuale*
Profling scambio di messaggi modello Tangent (TLROMS)		
Message Passage: 2D halo exchanges	565.420	0.5590 %
Message Passage: 3D halo exchanges	371.540	0.3673 %
Message Passage: 4D halo exchanges	117.870	0.1165 %
Message Passage: lateral boundary exchanges	12.840	0.0127 %
Message Passage: data broadcast	3703.245	3.6614 %
Message Passage: data reduction	3.585	0.0035 %
Message Passage: data gathering	59.775	0.0591 %
Message Passage: data scattering	424.930	0.4201 %
Message Passage: point data gathering	6.155	0.0061 %
Totale	5265.360	5.2059 %

Tabella 4.5: Output di ROMS-WC12 relativo al tempo impiegato nello scambio di messaggi

Frazione di sorgente	*Tempo*	*Percentuale*
Profling scambio di messaggi modello Adjoint (ADROMS)		
Message Passage: 2D halo exchanges	492.595	0.3574 %
Message Passage: 3D halo exchanges	361.500	0.3673 %
Message Passage: 4D halo exchanges	147.430	0.1458 %
Message Passage: lateral boundary exchanges	16.020	0.0158 %
Message Passage: data broadcast	1737.850	1.7182 %
Message Passage: data reduction	41.845	0.0414 %
Message Passage: data gathering	29.090	0.0288 %
Message Passage: data scattering	58.265	0.0576 %
Message Passage: point data gathering	3.045	0.0030 %
Totale	2887.640	2.8550 %

Tabella 4.6: Output di ROMS-WC12 relativo al tempo impiegato nello scambio di messaggi

ROMS pesa 5.2%, mentre la parte relativa al modello **ADROMS** pesa il 2.8% per cui le cause della scarsa scalabilità sono da ricercare in altre sezioni del software.

Per stabilire i bound dell'architettura sottostante, come osservato nel

#proc	*Tiling*	*Sostenuta*	*Picco*	*T(n) secondi*	*Speed-Up*	*Efficienza*
1	1x1	0.0125	3,4	75508.305	1	1
2	2x1	0.029	6,8	47112.885	1.6	0.8
4	2x2	0.0355	13,6	32128.285	2.34	0.58
16	4x4	0.0251	54,4	17547.515	4.3	0.26
36	6x6	0.0275	122,4	22879.365	3.3	0.09
64	8x8	0.0175	217,6	21478.695	3.51	0.05
100	10x10	0.0167	340	21572.515	3.5	0.035
144	12x12	0.012	489,6	22062.095	3.42	0.023

Tabella 4.7: Tempi e flops ottenuti con il test case WC12

paragrafo 2.2.1 la composizione del grafico consiste nell'ottenere i limiti di memoria e i limiti computazionali. Per il picco relativo alla picco di prestazione computazionale è sufficiente utilizzare il P_{peak}, mentre B_{peak} si ottiene mediante strumenti di micro-benchmarking in questo caso è stato utilizzato STREAM descritto nel paragrafo 2.2.2 poiché largamente utilizzato dalla comunità scientifica e perchè fornisce una stima affidabile di questo limite

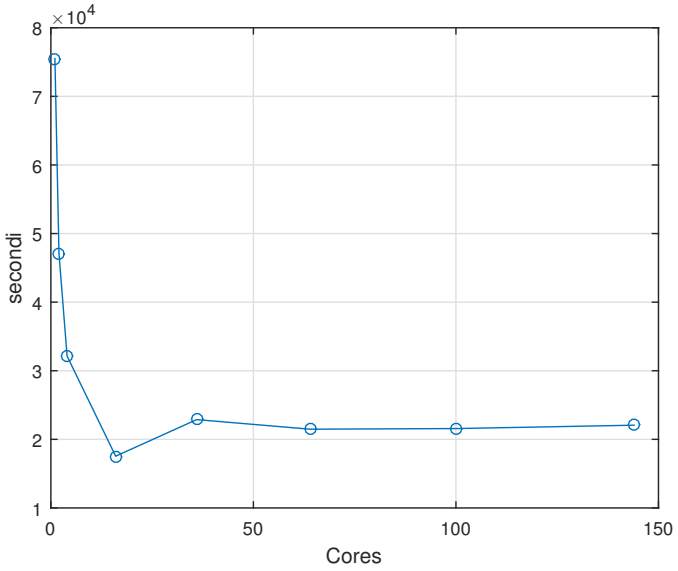


Figura 4.11: Tempi ottenuti per il test case WC12

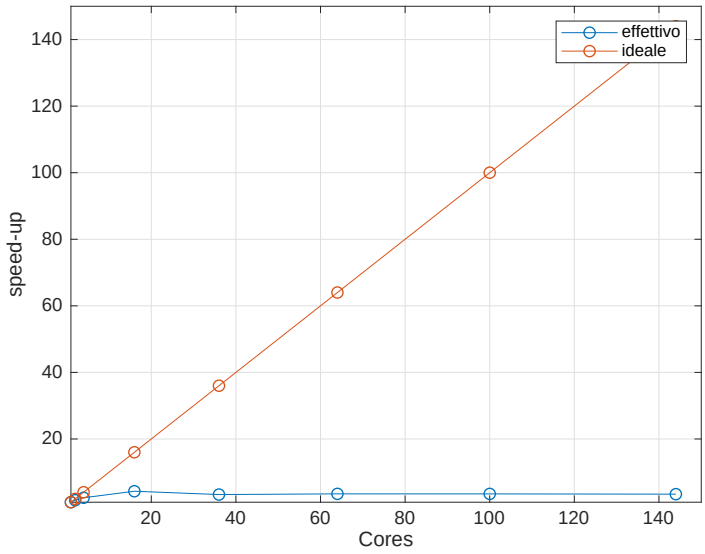


Figura 4.12: Speed up relativi ai tempi su WC12

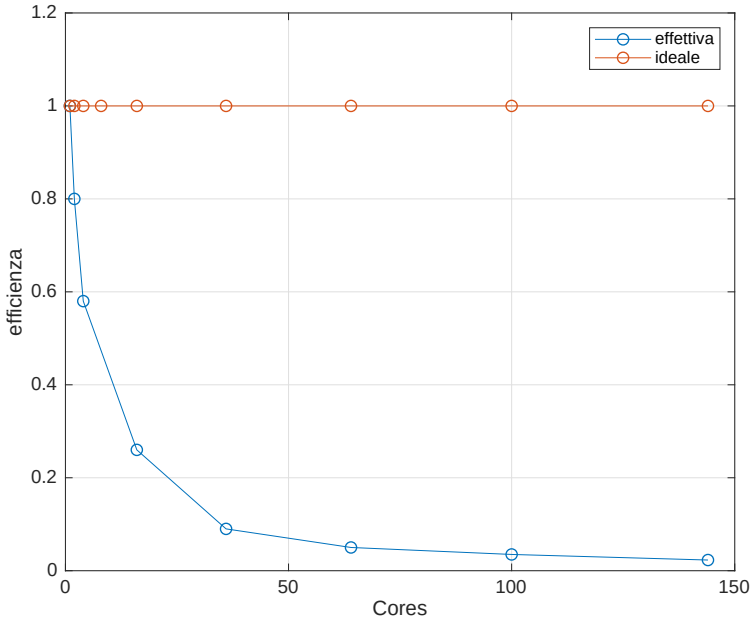


Figura 4.13: Efficienza relativa ai tempi su WC12

[26]. I nodi del Mont-Blanc hanno una memoria RAM di tipo *LPDDR3* da 4 GB come osservato in 1.2. La LPDDR3 è una speciale memoria ram a basso consumo utilizzata principalmente in smartphone e tablet.
Una volta ottenuti i due valori relativi alla potenza computazionale di picco e quella relativa alla memoria è possibile tracciare i due bound (definiti in letteratura anche come *ceilings*, Figura 4.14). Il punto di incontro delle due

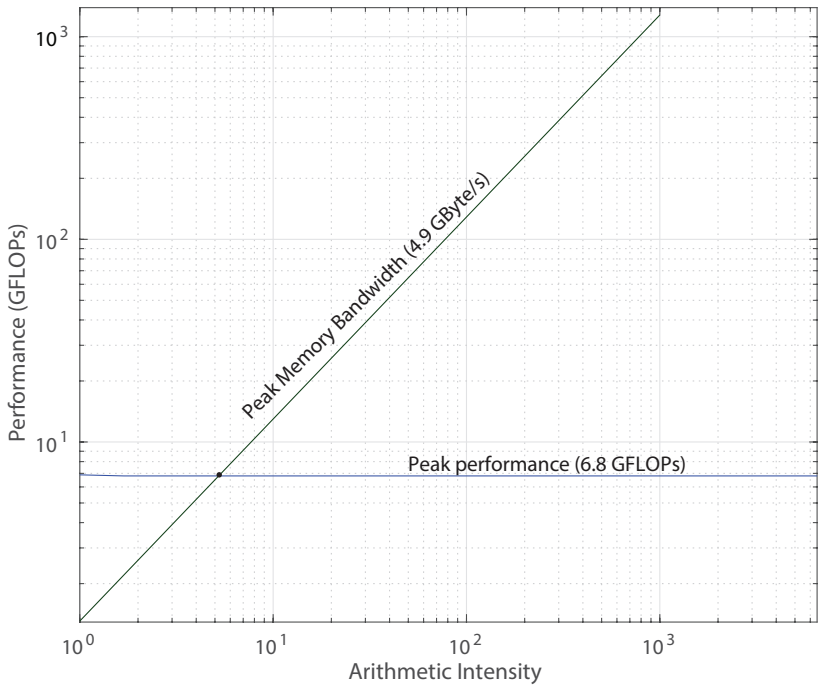


Figura 4.14: I due limiti computazionali dell'architettura ARM

rette da luogo a quello che roofline definisce come *ridge point* che rappresenta il punto in cui si ha la minima intensità aritmetica (definita in 3.5) al massimo delle prestazioni computazionali ottenibili, esso può guidare il programmatore ad uno sviluppo del software orientato all'ottenimento della massima efficienza.
Graficamente la retta relativa al bandwidth di memoria di picco del nucleo computazionale di TRIAD (come osservato in 2.6) è rappresentata dalla seguente equazione.

$$GFLOPs = 4.941(GBytes/s) * AI \tag{4.1}$$

e nel grafico è una retta obliqua, che è limitata superiormente dalla retta:

$$y = 6.8 \tag{4.2}$$

ovvero il picco computazionale P_{peak}.
Da notare il *ridge point*, detto anche punto di rottura, ovvero quel punto di confine in cui un software può essere limitato o dalla memoria o dall'intensità aritmetica, viene calcolato in questo modo:

$$rp = \frac{P_{peak}}{B_{peak}} = \frac{6.8}{5.9} = 1.15 \tag{4.3}$$

dove P_{peak} è la prestazione di picco massima e B_{peak} è il picco del bandwidth di memoria. Per cui il *ridge point* (Figura 4.15) ha coordinate:

$$(1.15, 6.8)$$

Il posizionamento dei kernel del software monitorato all'interno del grafico da una indicazione sul possibile miglioramento di cui essi avrebbero bisogno, e spinge il programmatore a capire se il nucleo è limitato dalla memoria (kernel alla sinistra del ridge point) oppure raggiunge dalla potenza computazionale o da entrambi.
E' possibile anche aggiungere ulteriori limiti al grafico per meglio dettagliare il comportamento del software sull'architettura in uso. Il processore ARM A15 su cui sono stati effettuati i test mette a disposizone [43] il supporto al *VPFv4* che utilizza l'FMA (fused multiply-add) ed esegue una addizione ed una di moltiplicazione contemporaneamente. Il picco computazionale utilizzato in roofline tiene in considerazione anche il mancato utilizzo dell'FMA per questo motivo al di sotto di questo limite il processore esegue una sola operazione (addizione o moltiplicazione) alla volta. Ottenuto in questo modo (Figura 4.16):

$$2\ core * 1.7\ GHz * 1\ op/cy = 3.4\ GFLOPs \tag{4.4}$$

Un ulteriore limite da inserire nel grafico è quello che rappresenta il mancato utilizzo del multi-core, per cui si considera l'utilizzo di un singolo core per volta, il risultato in questo caso resta il medesimo ottenuto per il limite del precedente caso quindi si ha una retta pari a $3.4(GFLOPs)$. Un ulteriore bound può essere inserito considerando il mancato utilizzo della FMA su solo core, dato che ciascun core potrebbe utilizzare il parallelismo a livello di istruzioni. Quindi il nuovo bound è così ottenuto:

$$1\ \text{core} * 1.7\ \text{GHz} * 1\ \text{op/cy} = 1.7\ \text{GFLOPs} \tag{4.5}$$

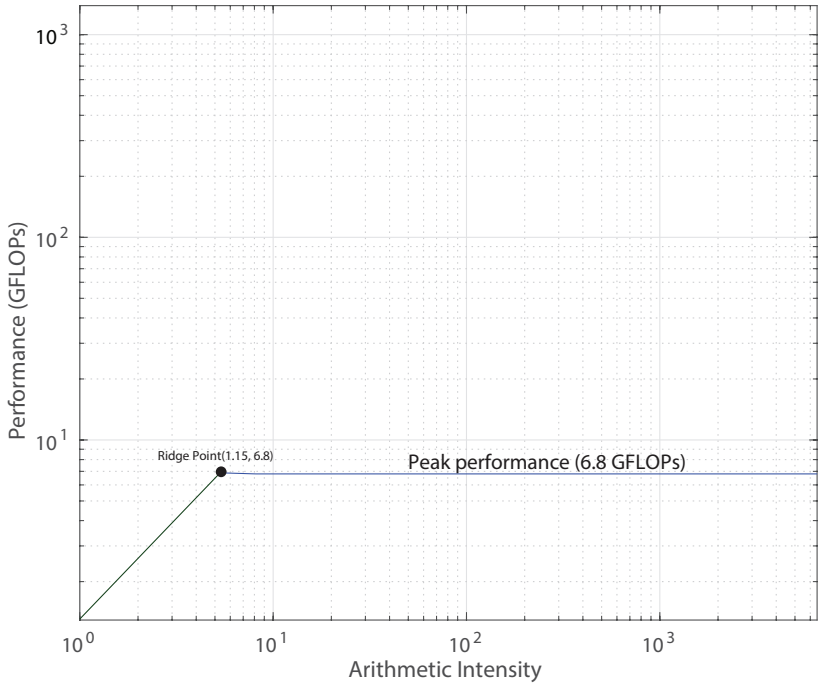


Figura 4.15: Ridge Point per ARM A15

Il grafico risultante costruito con i limiti appena illustrati ha la composizione in Figura 4.16, il grafico rappresentante tutti i limiti misurati è in Figura 4.17. La prossima fase si concentrerà sull'analisi di alcuni dei nuclei

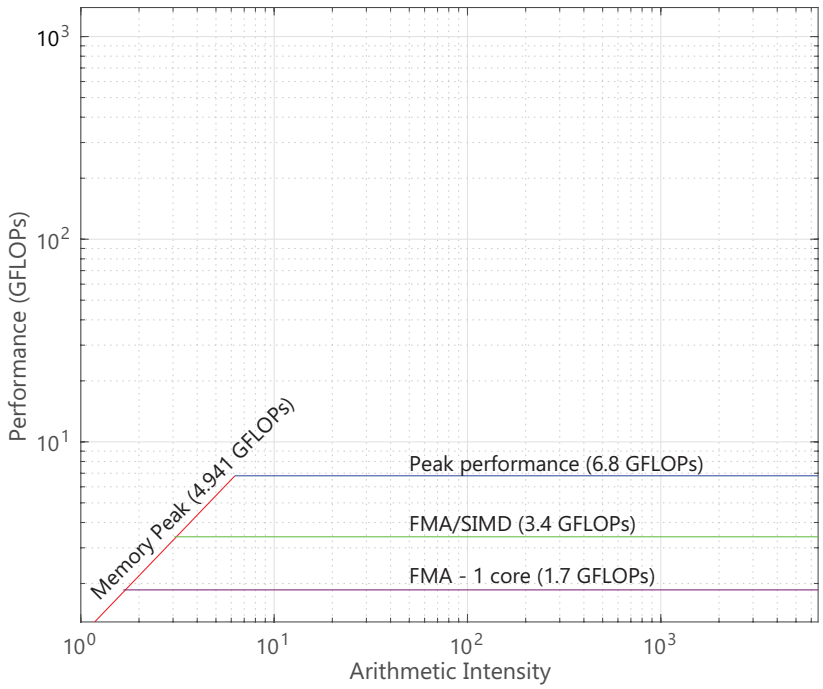


Figura 4.16: grafico Roofline con limiti computazionale e limite di memoria

computazionali più onerosi per classificarli all'interno del grafico roofline in modo tale da capire dove viene impiegata la maggior parte del tempo.

4.4.1 Analisi dei colli di bottiglia

Stabiliti i due limiti si procede a monitorare i nuclei computazionali che occupano la maggior parte del tempo di esecuzione. Il software ROMS viene analizzato con il tool GPROF (3.4) per evidenziare i nuclei che richiedono un maggiore sforzo computazionale. I risultati ottenuti con gprof sono riportati in 4.1.

Listing 4.1: Output di gprof per il modulo IS4DVAR di ROMS

```
Flat profile:
```

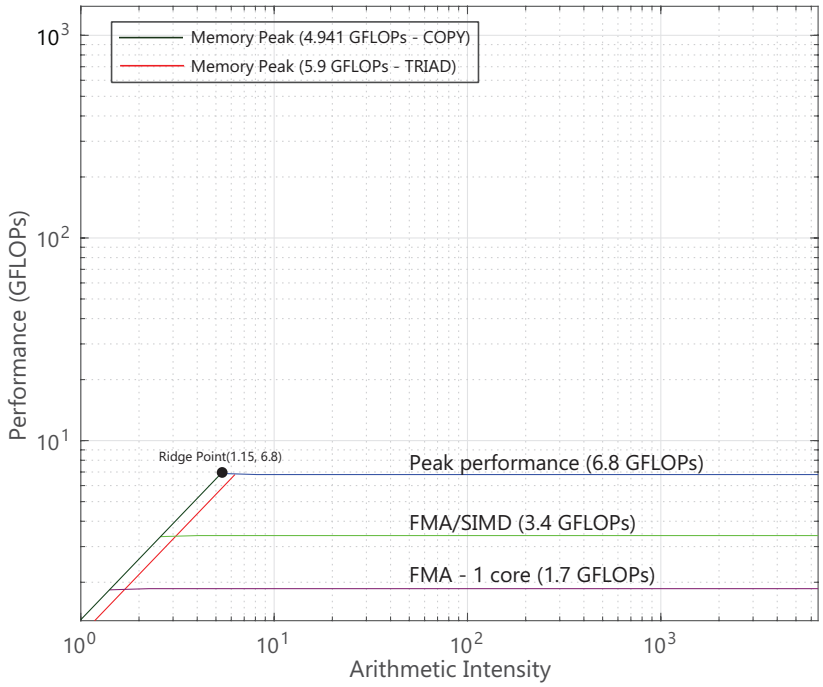


Figura 4.17: grafico Roofline con limiti computazionale e limite di memoria

```
Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  Ks/call  Ks/call  name
 16.15     398.04   398.04  1929024     0.00     0.00
    ad_step2d_mod_ad_step2d_tile_
 12.12     696.82   298.78  1929024     0.00     0.00
    tl_step2d_mod_tl_step2d_tile_
  6.14     848.16   151.35     9843     0.00     0.00
    ad_t3dmix_mod_ad_t3dmix2_tile_
  6.00     996.02   147.85     9843     0.00     0.00
    ad_pre_step3d_mod_ad_pre_step3d_tile_
  3.96    1093.59    97.57     9792     0.00     0.00
    tl_pre_step3d_mod_tl_pre_step3d_tile_
  3.32    1175.35    81.76     9792     0.00     0.00
    ad_step3d_t_mod_ad_step3d_t_tile_
  2.89    1246.60    71.25   127959     0.00     0.00
    set_3dfldr_mod_set_3dfldr_tile_
```

```
2.59    1310.57     63.97      9792      0.00      0.00
    tl_step3d_t_mod_tl_step3d_t_tile_
2.57    1373.85     63.28    127959      0.00      0.00
    set_3dfld_mod_set_3dfld_tile_
2.38    1432.51     58.66      9792      0.00      0.00
    ad_step3d_uv_mod_ad_step3d_uv_tile_
2.22    1487.28     54.77      9843      0.00      0.00
    ad_prsgrd_mod_ad_prsgrd_tile_
2.17    1540.73     53.45      9843      0.00      0.00
    ad_rhs3d_mod_ad_rhs3d_tile_
2.07    1591.69     50.95      9792      0.00      0.00
    tl_step3d_uv_mod_tl_step3d_uv_tile_
1.76    1635.04     43.35      9792      0.00      0.00
    tl_t3dmix_mod_tl_t3dmix2_tile_
```

In tabella 4.8 è possibile osservare un sottoinsieme dell'output fornito da GPROF. Si è deciso di mostrare in forma grafica l'output fornito da gprof in modo da rendere più leggibile l'output testuale, mediante *gprof2dot* [44] in Figura 4.18.

Kernel	*%tempo*	*time*
___ad__step2d__mod__MOD__ad__step2d__tile	25.59%	398.04
___tl___step2d__mod__MOD__tl__step2d__tile	18.41%	298.78
___ad__t3dmix__mod__MOD__ad__t3dmix2	8.00%	151.35

Tabella 4.8: Dati di profiling per i tre kernel computazionalmente più onerosi

La routine *ad_step2d_tile* (Figura 4.19) è quella che occupa la maggior parte del tempo in ROMS con il 15% del costo totale dell'intera esecuzione. La seconda routine instrumentata che richiede maggiore tempo è *tl_step2d_tile* (Figura 4.20) che pesa il 14% sull'intera esecuzione di ROMS. In relazione a questo grafo si può osservare che le routine del modulo IS4DVAR di ROMS computazionalmente più onerose sono quelle relative ai due passi *Tangent Linear* ed *Adjoint* dell'algoritmo in Figura 4.5. Dato che l'intera esecuzione coinvolge per lo più queste due routine è utile instrumentarle, è studiarne le prestazioni in termini di prestazione sostenuta (GFLOPs) ed accessi alla memoria.
Per posizionare i kernel più onerosi all'intero del grafico del del modello roofline sono necessari i seguenti tre valori:

- Il numero di operazioni floating point di un kernel N_{flop};
- Il tempo di esecuzione di un kernel $T_{sw}(A)$;

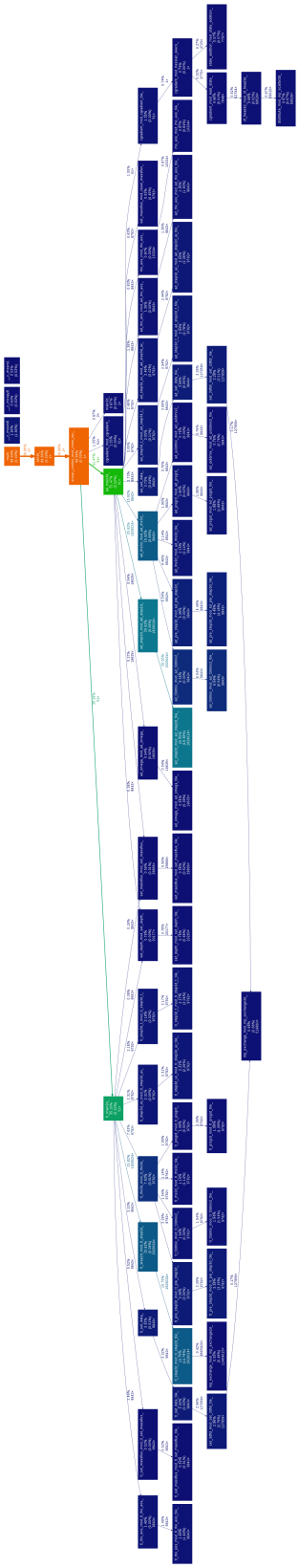


Figura 4.18: Output in forma grafica di gprof che mostra in colori diversi le routine più onerose

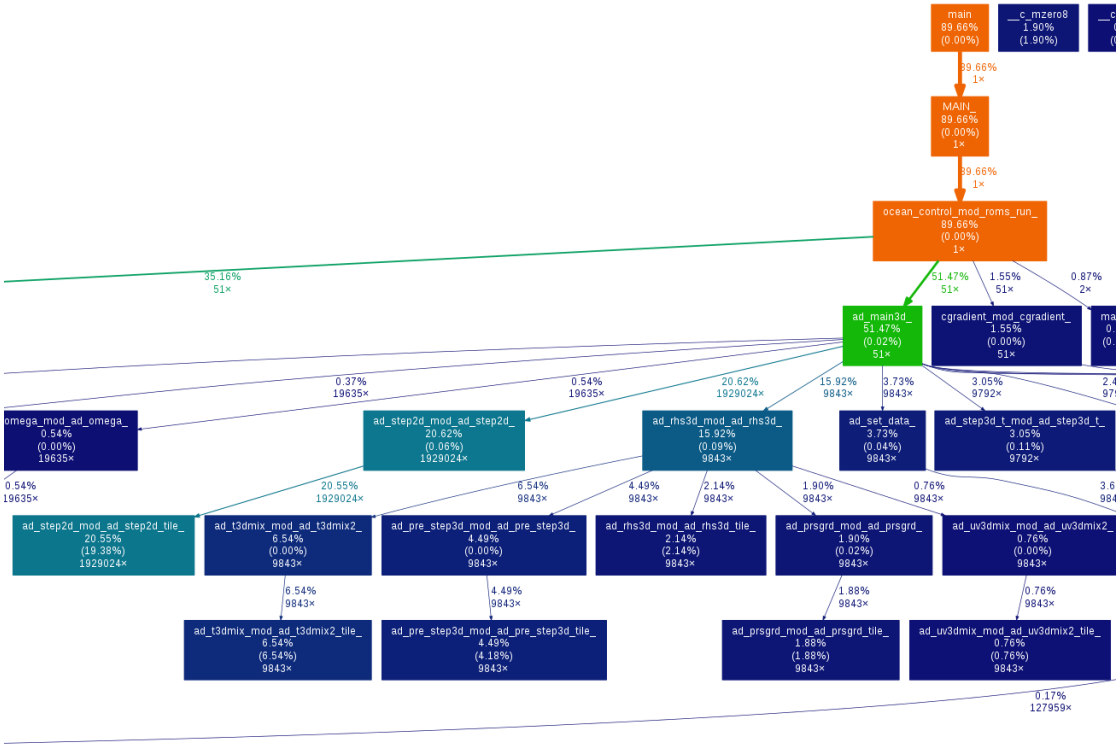


Figura 4.19: ingrandimento dell'output di gprof relativo ad una della due routine di ROMS computazionalmente più onerose

- Il totale dei dati trasferiti da e per la memoria principale N_{mem}.

Per ottenere la prestazioni sostenuta, come già mostrato, si divide il numero di operazioni di un kernel per il tempo di esecuzione dello stesso (definita in 1.5). L'intensità aritmetica (AI, definita in 3.5) si ottiene dividendo il numero di bytes trasferiti da e per la memoria per il numero totale di FLOPS (ovvero l'ascissa del roofline chart). Per completare la tabella della roofline chart è stata utilizzata la libreria PAPI descritta nel paragrafo 3.1. Il numero di FLOPS viene estratto utilizzando la metrica *PAPI_FP_INS* quindi:

$$P_{actual} = \frac{PAPI\_FP\_INS}{T_{sw}(A)} \tag{4.6}$$

dove $T_{sw}(A)$ è il tempo di esecuzione.
Per ottenere il bandwidth di memoria (in GByte/s) si ricorre alla metrica derivata relativa ai cache miss della cache L2 (PAPI_L2_DCM). Il numero di

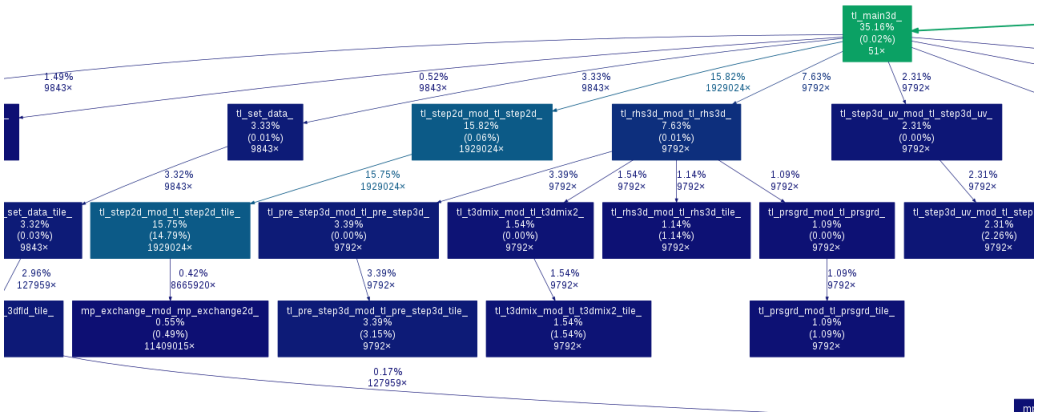


Figura 4.20: ingrandimento dell'output di gprof relativo alla rotuine tl_step2d_tile di ROMS che risulta essere computazionalmente onerosa

cache missdella cache di livello L2 (il livello più alto disponibile sull'architettura in uso) implicano che il dato venga poi prelevato dalla memoria centrale per questo la misura per cui si può considerare la metrica affidabile.Quindi:

$$B_{actual} = \frac{PAPI\_L2\_DCM}{T_{sw}(A)} \tag{4.7}$$

In tabella 4.9 ci sono i dati raccolti per i due nuclei, si trova il tempo di esecuzione totale di ciascuna funzione, la performance sostenuta denominata con P_{actual}, il numero di cache miss relativi alla cache di livello L2, l'intensità aritmetica che viene calcolata come definito in 3.5, ovvero il rapporto tra il numero di operazioni floating-point e al numero di accessi alla memoria. In Figura 4.21 si osserva la funzione *ad_step2d_* dove ciascun punto rappresenta una diversa configurazione di esecuzione, con un tiling pari a 2×1, la routine ha una bassa intensità aritmetica e non si avvicina al limite di memoria ciò vuol dire che ha una bassa richiesta di dati dalla memoria rispetto alla quantità di operazioni. Per la configurazione 2×2 si nota come la routine faccia un uso maggiore della memoria e l'intensità aritmetica sia molto maggiore, questo succede probabilmente poiché si ha un numero superiore di dati da aggregare. Il *ridge point* ha coordinate $(1.15, 6.8)$ ed osservando l'esecuzione 2×1 il punto relativo a questa configurazione è a

sinistra del ridge point, ciò significa che in questa configurazione in software non è ulteriormente migliorabile a causa del memory bound, questo vuol dire che si è molto vicini al limite imposto dal bandwith della memoria.
La seconda funziona presa in considerazione è quella relativa al modulo

Kernel	*Tiling*	$T_{sw}(A)$	P_{actual}	B_{actual}	AI
	2×1	0.0138	0.051	0.00120	0.57
	2×2	0.0233	0.156	0.0024	1.5
ad_step_2d_tile	8×8	0.0213	0.1	0.0001	1.38
	2×1	0.0414	0.149	0.00456	1.35
	2×2	0.0275	0.111	0.0019	1.55
tl_step_2d_tile	8×8	0.48	0.000039	0.00012	1.56

Tabella 4.9: Dati relativi alla costruzione del roofline model

del *tangent linear* di IS4DVAR. In Figura (4.22) sono stati monitorate tre configurazioni di tiling differenti ovvero 2×1,2×2,8×8. La linea obliqua in rosso rappresenta il memory bound e quella blu rappresenta il computational bound. Le due configurazioni su cui porre l'attenzione sono quelle relative al 8×8 e 2×2 poiché tra le due c'è decadimento delle prestazioni all'aumentare il numero di processori, nonostante l'intensità aritmetica sia quasi la stessa, questo ci indirizza verso l'idea che il software all'aumentare del numero di processori abbassi troppo l'efficienza senza ottenere un reale guadagno. Le possibili motivazioni per cui ROMS mostra questo comportamento potrebbe essere legato a molteplici cause. A partire al fatto che il software è stato testato sul cluster Mont-Blanc costituito da CPU molto poco potenti, cioè che hanno una prestazione di picco molto bassa ma che in compenso offrono bassi consumi, potrebbe essere interessante quindi l'utilizzo di una architettura di questa tipologia quando è richiesta l'utilizzo di un software simile con consumi e costi bassi.
Inoltre c'è da considerare la dimensione della cache a disposizione sul Cortex ARM-A15 (fino a 4MB) che essendo di dimensione esigua aumenta la probabilità di riferimenti alla memoria principale in caso di cache miss.
Alcune delle azioni da compiere per migliorare le prestazioni del software richiedono un re-design del codice, ovvero una riscrittura in base all'architettura target su cui ROMS deve essere eseguito, ad esempio:

- *FMA balance*: Dai dati raccolti non è presente un buon bilanciamento di addizioni e moltiplicazioni ciò non permette di sfruttare appieno l'FMA, ovvero la capacità del processore di eseguire una moltiplicazione ed una addizione contemporaneamente;

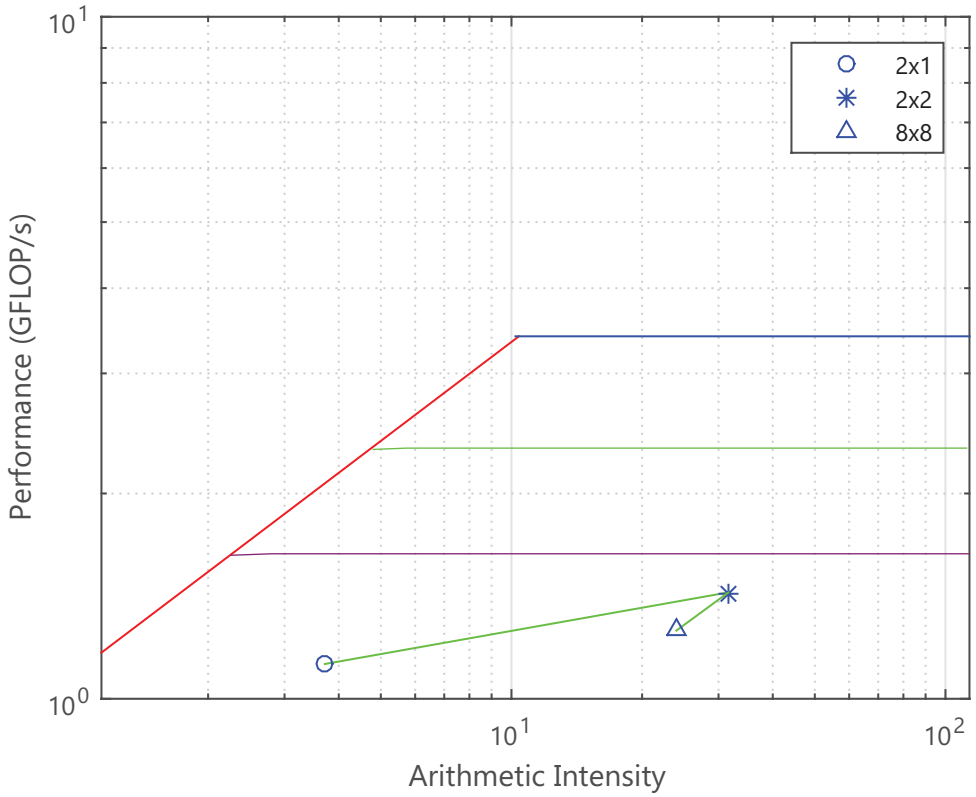


Figura 4.21: Risultati ottenuti per la routine di ROMS ad_step_2d_tile

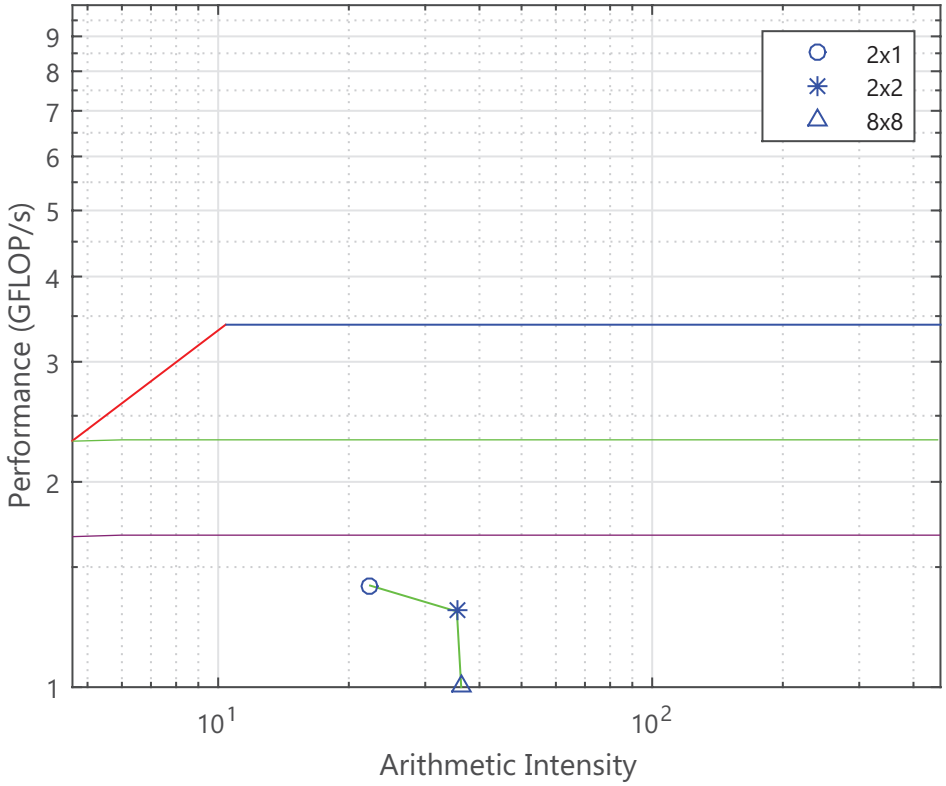


Figura 4.22: Risultati ottenuti per la routine di ROMS tl_step_2d_tile

- Per aumentare le prestazioni si potrebbe anche riscrivere il codice in maniera tale da favorire il *software prefetching* così da portare nelle memorie veloci i dati prima che essi siano necessari, anche in questo caso è obbligatorio un re-design del software;
- Inoltre potrebbe essere utile permettere una esecuzione ibrida del software, possibilità al momento negata dagli sviluppatori, per sfruttare al meglio le possibilità offerte sia dal calcolo intra-nodo che da quello inter-nodo.

Capitolo 5

Conclusioni

Questo lavoro di tesi va a collocarsi nel campo del calcolo parallelo, in particolare occupandosi dello studio delle metodologie e delle tecniche per valutare le prestazioni di software parallelo. La prestazione di un calcolatore è valutata in termini di **tempo di risposta** quindi si sono definiti i concetti di prestazione sostenuta e prestazione di picco, ed il confronto tra le due quantità. Utilizzando **benchmark** per valutare il sistema di calcolo su problemi noti, ed utilizzando strumenti specifici per la valutazione delle prestazione del software. Il lavoro di tesi si prefigge di descrivere i parametri per valutare se una applicazione è adatta o meno ad essere eseguita su architetture ad alte prestazioni in particolare su architetture exascale in grado di eseguire un *Exaflop/s*. Per questo scopo si illustra dapprima il concetto di benchmark (3.1) e gli strumenti allo stato dell'arte per la classificazione delle architetture ad alte prestazioni, come **Linpack Benchmark**, **HPL** e **HPCG** che forniscono una metrica univoca per la classificazione di un sistema di calcolo in termini di **FLOPs**. Nel paragrafo 2.2.1 è stato illustrato il benchmark **STREAM** utile a valutare l'ampiezza del bandwidth di memoria del sistema di calcolo. Nel capitolo 2.2.1 sono stati eseguiti sui sistemi di calcolo a disposizione i benchmark utili per la valutazione del caso studio oggetto di questa tesi: il software ROMS, in Figura 5.1. Nel capitolo 3.1 sono stati illustrati gli strumenti collezionano informazioni sulla valutazione delle prestazioni, in Figura 5.2 è possibile osservare graficamente le relazioni tra gli strumenti per la valutazione del software. Generalmente l'iter per la valutazione delle prestazioni di un software consta principalmente di questi passi:

- Analisi delle prestazioni del codice sorgente in termini di strong scalability (effettuata mediante la tecnica dell'interval timing);

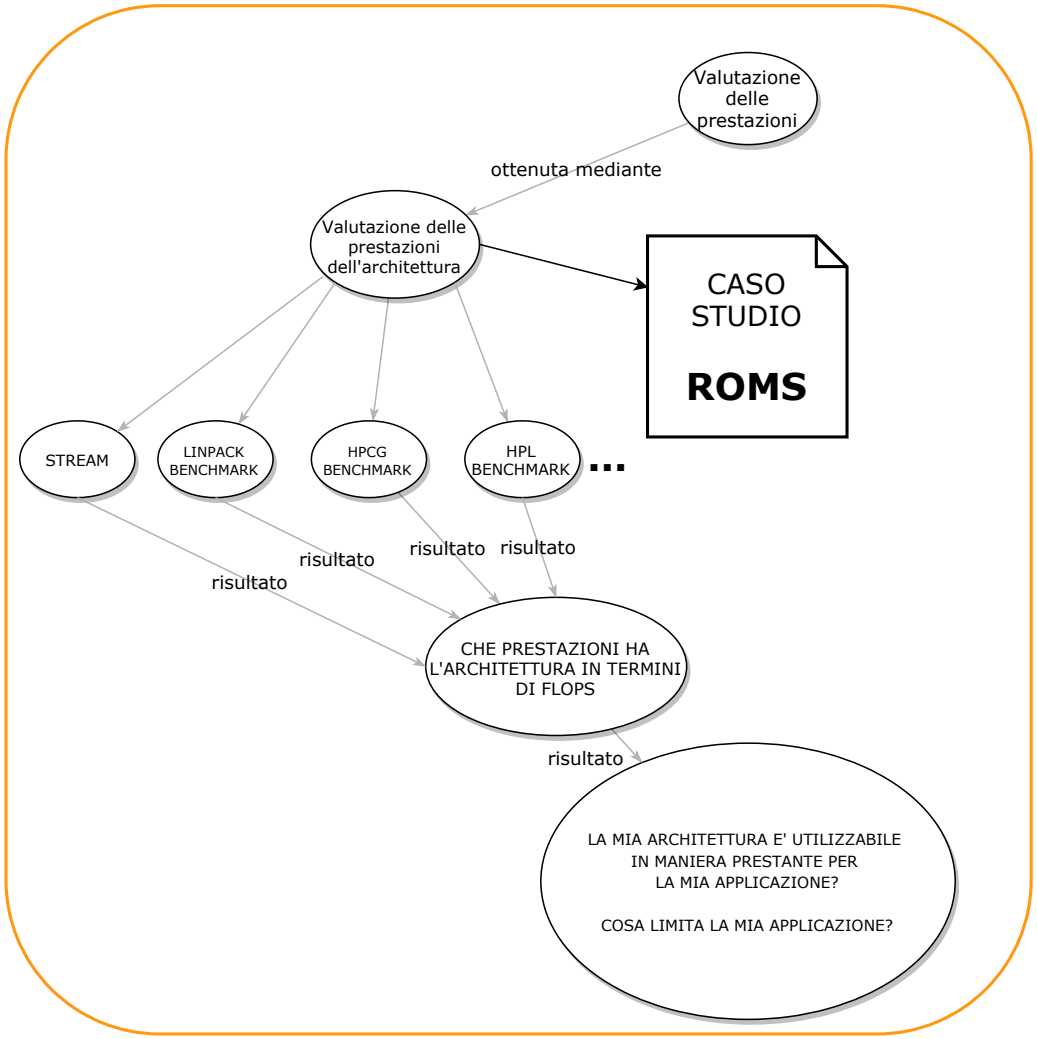


Figura 5.1: Gli strumenti utili al calcolo della prestazione di picco e di quella sostenuta su architetture HPC effettuati per questo lavoro di tesi

- Analisi dei nuclei computazionali onerosi (effettuata con GPROF)
- Tracciamento degli eventi (effettuato con PAPI)

il primo passo può essere portato a termine eseguendo il software in oggetto e analizzando i tempi di esecuzione, utili al calcolo dello speed-up e dell'efficienza del codice parallelo. Per il secondo passo sono stati usati gli strumenti del capitolo 3.1, ovvero **PAPI** e **GPROF**, il primo fornisce una **API** per accedere ai contatori hardware utilizzati durante l'esecuzione, il secondo permette di ottenere una lista dei nuclei computazionali più onerosi. Infine i dati raccolti vanno a comporre il **modello roofline** basato principalmente su due fattori:

- Numero di operazioni floating point N_{flop};
- Numero di accessi alla memoria N_{mem}.

utilizzando tutte le informazioni messe a disposizione dagli strumenti descritti e mediante l'uso di modelli per le prestazioni si può trarre un'analisi che suggerisca come il software possa essere eseguito in maniera efficiente su architetture ad alte prestazioni.
Software come **ROMS**, basati sul clima utilizzano una grande mole di dati, ed effettuano una grande numero di computazioni tra i sottodomini, questo pone l'attenzione sull'importanza del raggiungimento di una alta efficienza, ovvero l'applicazione deve essere in grado di sfruttare appieno le caratteristiche dell'architettura sottostante.

Alla luce di queste analisi sono stati fatti dei test a partire da un unico dominio e poi via via suddividendolo in parti più piccole, da queste questa analisi si evince che l'applicazione ha una bassa efficienza, quindi che non è ben predisposta a sistemi di calcolo di nuova generazione. In Figura 5.3 si possono osservare i tempi di esecuzione mentre in Figura 5.4 si nota la bassa efficienza anche solo utilizzando più di due nodi. Considerando anche l'analisi roofline che pone l'attenzione sui nuclei computazionali onerosi, osservabili in Figura 5.5, e 5.6 si può dire che ROMS nelle sue routine computazionalmente più onerose raggiunge soltando lo 0.01% del picco di prestazioni. Un modo per migliorare la prestazione, può essere fatto sfruttando alcune cattatteristiche di ottimizzazione dei compilatori come la *loop vectorisation*[45]. L'analisi portata a termine indirizza ad una riscrittura del software ed in particolare:

- *FMA balance*: Dai dati raccolti non è presente un buon bilanciamento di addizioni e moltiplicazioni ciò non permette di sfruttare appieno l'F-

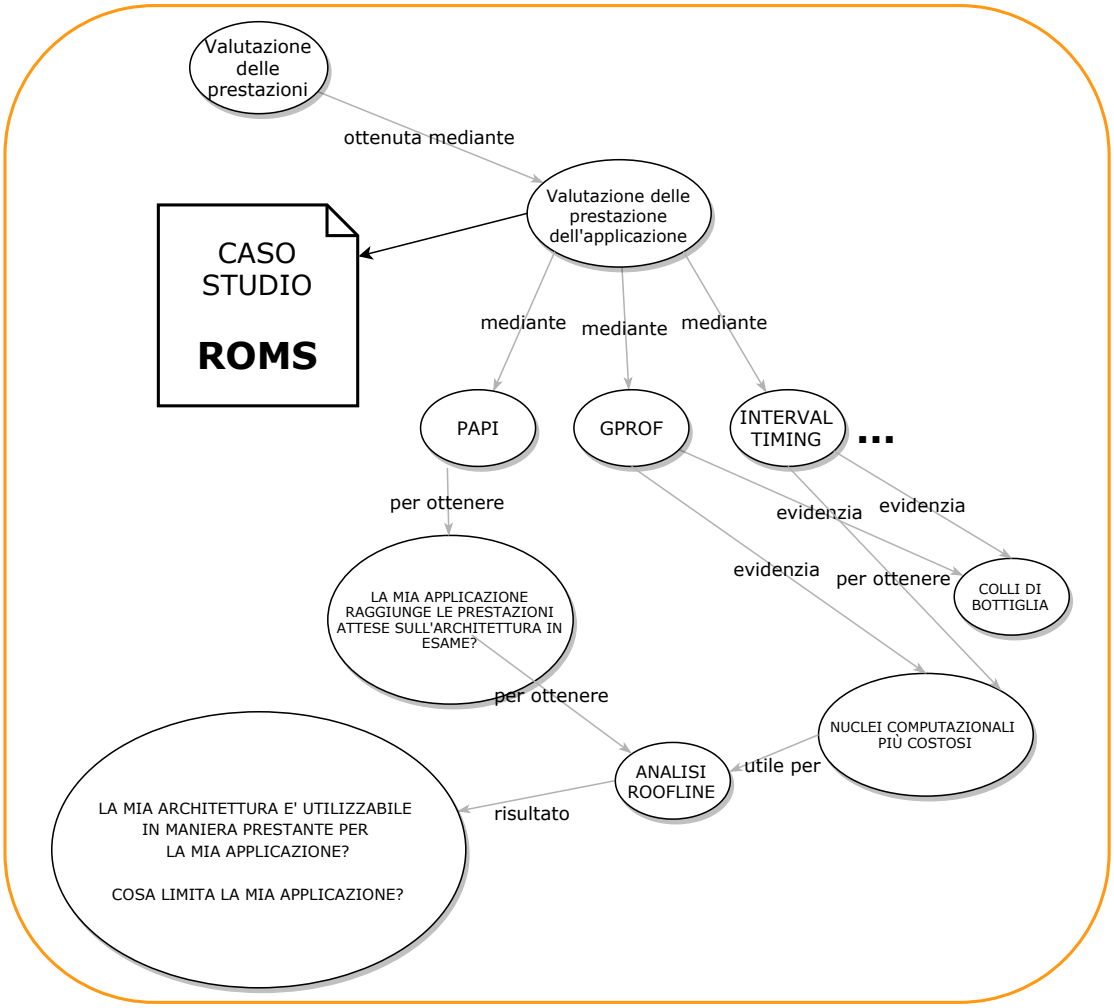


Figura 5.2: Gli strumenti utili al calcolo della prestazione di software parallelo

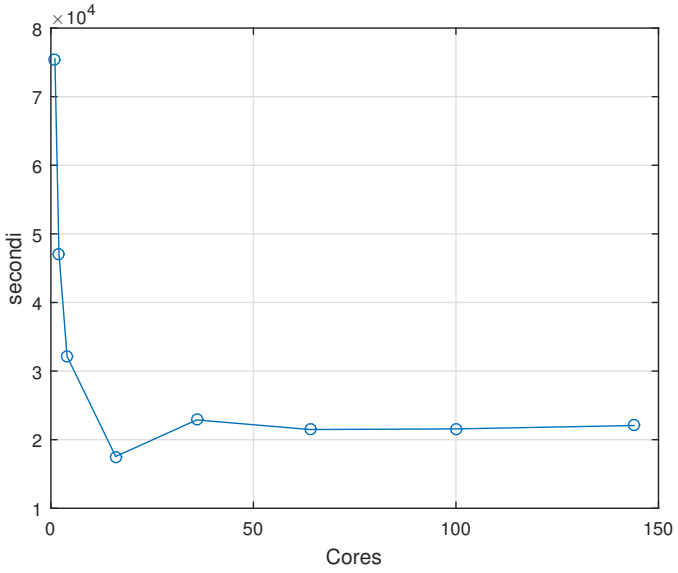


Figura 5.3: Tempi su WC12

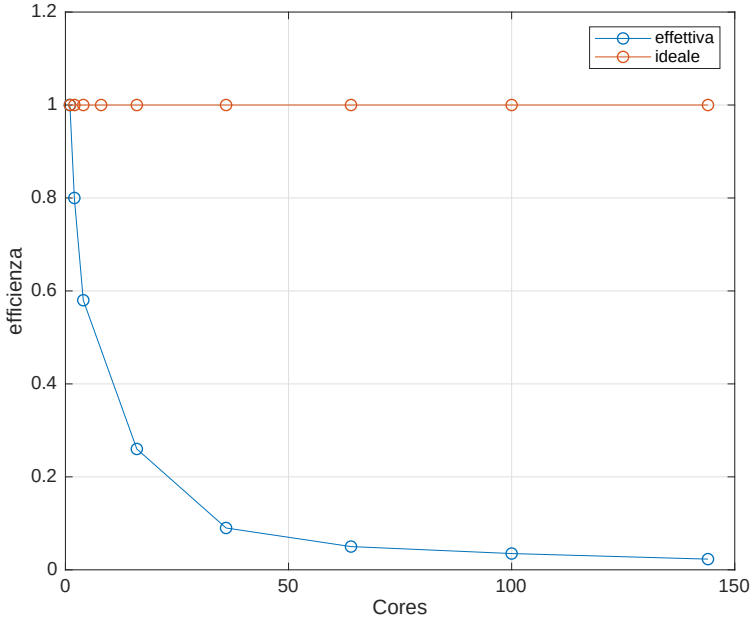


Figura 5.4: Efficienza su WC12

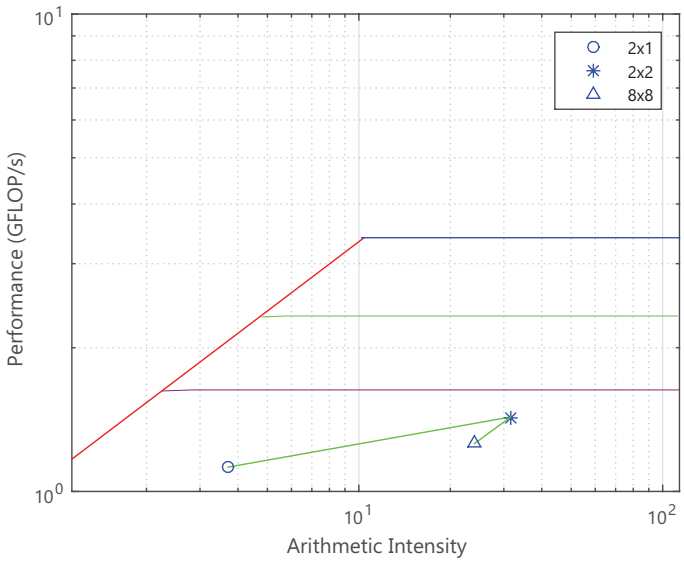


Figura 5.5: Analisi roofline sulla funzione ad_step_2d

MA, ovvero la capacità del processore di eseguire una moltiplicazione ed una addizione contemporaneamente;

- Per aumentare le prestazioni si potrebbe anche riscrivere il codice in maniera tale da favorire il *software prefetching* così da portare nelle memorie veloci i dati prima che essi siano necessari, anche in questo caso è obbligatorio un re-design del software;

- Inoltre potrebbe essere utile permettere una esecuzione ibrida del software, possibilità al momento negata dagli sviluppatori, per sfruttare al meglio le possibilità offerte sia dal calcolo intra-nodo che da quello inter-nodo.

Come è evidente la valutazione delle prestazioni non è di semplice lettura e richiede un'analisi molto approfondita delle architetture per capire come esse possano essere sfruttate al meglio, considerando anche la continua evoluzione a cui sono soggette. Per ottenere una buona analisi c'è la necessità di utilizzare strumenti di diversa natura e complessi nella loro comprensione, questo lavoro si pone l'obiettivo di rappresentare una guida per meglio comprendere come essi possano essere usati congiuntamente.

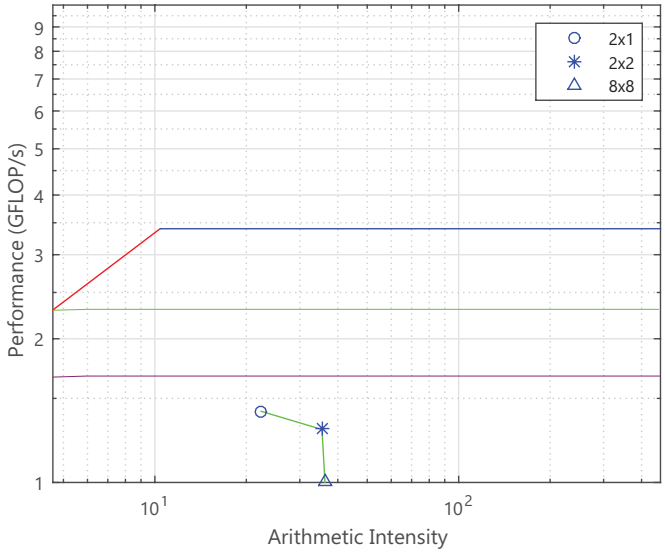


Figura 5.6: Analisi roofline sulla funzione tl_step_2d

Appendici

Appendice A

Calcolo della prestazione sostenuta per algoritmi noti

A.0.1 Calcolo della prestazione sostenuta mediante un esempio

Si vuole fornire un esempio pratico di come sia possibile calcolare la prestazione sostenuta manualmente per meglio comprendere il comportamento del processore. Si supponga di voler calcolare il numero di operazioni compiute in 2 minuti data la prestazione di picco del processore Intel $P_{peak} = 23,2\ GFLOPs$, otterremo:

$$ops = 120\ secondi \times 23.200.000.000\ Flops = 2.796.000.000.000 = 2,79 \times 10^{12} \tag{A.1}$$

quindi il massimo n teorico calcolabile in 2 minuti è $2,79^{12}$. Per verificare effettivamente che ciò sia vero è necessario implementare l'algoritmo e prendere i tempi di esecuzione.
L'algoritmo in pseudocodice per la somma di n numeri:

Pseudocodice

```
program somma_n_numeri(in: n; out: sum)
begin
  var: n, sum: integer
  var: a(n) : real

  read(n)
  for i = 1 to n do
        sum = sum + a(i)
  endfor
```

```
end
```

di seguito l'algoritmo in C:
Codice Sorgente

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <float.h>

/*====================
   SOMMA +a -a
  ====================*/

/* compilazione con livello di ottimizzazione 1
 * gcc -o somma_a-a -O1 somma_a-a.c
 */

/* output su Intel I5-3470S valutando anche l'assegnazione
 * duration=0.677874
 * MFLOPS: 5900.801624
 * La PEAK PERFORMANCE del processore I5-3470S e': 11.600000
 */

/* output su Intel I5-3470S non valutando l'assegnazione
 * duration=0.681032
 * MFLOPS: 2936.719567
 * La PEAK PERFORMANCE del processore I5-3470S e': 11.600000
 */
#define fpo 0 //commentare questa riga (e decommentare la
    successiva) per valutare solo le operazioni floating
    point
//#define fpo 1

int main()
{
  double somma=0.0,a=1.0;
  int i,n;
  float ops;
  struct timeval start,finish;
  double duration, flops;

  n = INT_MAX;

  gettimeofday(&start,NULL); //tempo inizio
  for (i=0; i<n;i++)
  {
          a = -a;
      somma += a;
```

```
  }
  gettimeofday(&finish,NULL); //tempo fine
  //raccolta tempo totale in microsecondi e conversione in
      milli
  duration=((double)(finish.tv_sec-start.tv_sec)*1000000+(
      double)(finish.tv_usec-start.tv_usec))/1000000;

  printf("\n\nduration=%lf\n",duration);
  if(fpo == 0) ops = 2 * 2e+9;
  else ops = 1 * 2e+9;
  flops= ops/duration*1.0e-6; //calcolo del mflops
  printf("MFLOPS: %f \n", flops);
  int nc = 4;
  float frequency = 2.9; //calcolo peak performance (per CPU
       intel)
  printf ("La PEAK PERFORMANCE del processore I5-3470S e': %
      f\n MFLOPS",nc * frequency);
    return 0;
}
```

La compilazione si effettua in questo modo:

```
$gcc -o nome_eseguibile -O1 nome_eseguibile.c
```

Nelle tabelle I tempi di esecuzione con e senza l'attivazione del livello di ottimizzazione 1 (per approfondimenti - Capitolo 5 [45]):

	Tempo medio (NOPT)	*GFLOPs*
100	0.000001	0.4
1000	0.000006	0.6
10.000	0.000058	0.69
100.000	0.000566	0.70
1.000.000	0.004660	0.85
10.000.000	0.028827	1.387
100.000.000	0.257179	1.555
1.000.000.000	2.550885	1.568
INT_MAX	5.485245	1.571

Tabella A.1: Calcolo della prestazione sostenuta senza livelli di ottimizzazione

Come è possibile osservare il numero dei MFLOPS cresce attivando tale opzione, ma si è ancora lontani dalla prestazione di picco.

	Tempo medio (O1)	*GFLOPs*
100	non valutabile	#
1000	0.000001	6.153
10.000	0.000006	6.400
100.000	0.000064	6.400
1.000.000	0.000661	6.051
10.000.000	0.005114	7.821
100.000.000	0.031802	12.577
1.000.000.000	0.285613	13.951
INT_MAX	0.609356	13.337

Tabella A.2: Calcolo della prestazione sostenuta attivando il livello 1 di ottimizzazione del compilatore

A.0.2 Caso Test: Prodotto Scalare

Scopo

Calcolo del prodotto scalare così definita y = x * y, con x e y vettori di reali.

Specifiche

Testata della routine - int dot_product(int *a, int *b, size_t n);

Descrizione

Dati x,y due vettori di interi di dimensione n, l'algoritmo implementato calcola il loro prodotto scalare eseguendo n volte la seguente operazione, e scrivendo il risultato parziale di una variabile d'appoggio sum (che conterrà il risultato finale:

$$sum = a(i) * b(i)$$

l'accesso al vettore avviene in maniera sequenziale in base all'indice corrente i. Parametri di INPUT:

- a - Array di dimensione n
- b - Array di dimensione n
- n - dimensione dei vettori

Parametri di OUTPUT:

- In output viene restituito un intero, risultato del prodotto scalare

Indicatori di errore

Se il numero di parametri non è corretto (ovvero non viene fornita la dimensione dei vettori in input) il programma uscirà con un errore (1).
Se il valore di ritorno è *EXIT_SUCCESS* la routine è terminata con successo

Routine ausiliarie richiamate

Nessuna

Tempo di Esecuzione

L'algoritmo implementato esegue N operazioni (1 moltiplicazione) floating-point.
Di seguito vengono forniti alcuni tempi di esecuzione dell'algoritmo:
Per l'ultima dimensione (ovvero 1 miliardo di elementi) i tempi non sono

Dimensione	*Durata (ms)*	P_{actual}	*Peak (GFLOPs)*
100	0.000001	0.200000	23.2
1000	0.000006	0.333333	23.2
10000 (10^4)	0.000071	0.281690	23.2
100000 (10^5)	0.000563	0.355240	23.2
1000000 (10^6)	0.005633	0.355051	23.2
10000000 (10^7)	0.042111	0.474935	23.2
100000000 (10^8)	0.254210	0.786751	23.2
	#	#	

Tabella A.3: DOT: Tempi di esecuzione e GFLOPs al variare della dimensione

disponibili perché la macchina non è in grado di eseguire l'algoritmo con quella dimensione (fare riferimento al prossimo paragrafo), il processo che esegue l'algoritmo viene trasferito nella memoria di SWAP quindi si considera n in funzione del quantitativo di memoria che si può sfruttare con un solo riempimento.
Utilizzando invece reali in doppia precisione questi sono i tempi di esecuzione ottenuti:
Come nella tabella precedente il massimo numero di elementi utilizzabili è inferiore a $5 * 10^8$, per una spiegazione dettagliata fare riferimento al paragrafo successivo sulla *memoria richiesta*.

Dimensione	*Durata (ms)*	P_{actual}	*Peak (GFLOPs)*
100	0.000001	0.200000	23.2
1000	0.000006	0.333333	23.2
10000 (10^4)	0.000057	0.350877	23.2
100000 (10^5)	0.000570	0.350877	23.2
1000000 (10^6)	0.005489	0.364365	23.2
10000000 (10^7)	0.040565	0.493036	23.2
100000000 (10^8)	0.258531	0.773602	23.2

Tabella A.4: DOT: Tempi di esecuzione e GFLOPs al variare della dimensione (in doppia precisione)

Memoria Richiesta

L'algoritmo implementato utilizza due array reali di dimensione N per la memorizzazione di x e y. Quindi la complessità di spazio è $O(n)$. Per ottenere la complessità di spazio utilizzata dall'algoritmo si fa ricorso alla formula 1.3.
Quindi nella funzione DOT ho 2 vettori di n elementi, per conoscere il numero di elementi che è possibile memorizzare in singola precisione sull'architettura Intel sarà sufficiente calcolare lo spazio in questo modo:

$$\frac{8.000.000.000Byte}{4} = 2.000.000.000 \tag{A.2}$$

quindi $n = 1.000.000.000$ poiché ho due vettori di dimensione n, dove 8.000.000.000 è la dimensione della RAM disponibile sul calcolatore (ricordando però che tale memoria non è totalmente utilizzabile a causa dei processi attivi del s.o.). Mentre in doppia precisione si ha

$$\frac{8.000.000.000Byte}{8} = 1.000.000.000 \tag{A.3}$$

quindi $n = 500.000.000$. Come dimostrato nel paragrafo precedente, effettivamente tali valori rappresentano il limite superiore che può assumere n.

Accuratezza Fornita

Per verificare l'accuratezza della routine sono stati inseriti valori numerici pari a 1.0 in modo tale che il risultato sia uguale ad N.

Esempio D'uso

Ecco come compilare il sorgente:
$gcc -o dot dot.c
Ecco come far girare l'eseguibile:
$./dot
Alcuni casi d'uso:

- ho due vettori: a = 1.0,1.0,1.0, b = 1.0,1.0,1.0. Lanciando l'eseguibile otterrò
  Risultato: 3.0
- ho due vettori: a = 2.0,1.0,1.0, b = 1.0,1.0,1.0. Lanciando l'eseguibile otterrò
  Risultato: 4.0
- ho due vettori: x = -1.0,1.0,1.0, y = 1.0,1.0,1.0. Lanciando l'eseguibile otterrò
  Risultato: 2.0

Pseudocodice

```
program prodotto_scalare(in: n; out: sum)
begin
  var: n, sum: integer
  var: a(n) : real

  for i = 1 to n do
        sum = sum + a(i) * b(i)
  endfor
end
```

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

/*** Scopo ***
Calcolo di un prodotto scalare del tipo result = ab, con x e
     y vettori di reali.

 *** Specifiche ***
Testata della routine dot_product - float dot_product(float
    *a, float *b, size_t n);

 *** Descrizione ***
```

```
Dati x,y due vettori di interi di dimensione n, l'algoritmo
    implementato calcola
il loro prodotto scalare eseguendo n volte la seguente
    operazione, e scrivendo il
risultato parziale di una variabile d'appoggio sum (che
    conterra' il risultato finale:
        sum = a(i) * b(i)
l'accesso al vettore avviene in maniera sequenziale in base
    all'indice corrente i.

 *** Riferimenti Bibliografici ***
Golub, Van Loan - Matrix Computations (Third Ed.)

 *** Parametri di input/output ***
Parametri di INPUT:
a - Array di dimensione n
b - Array di dimensione n
n - dimensione dei vettori
Parametri di OUTPUT:
In output viene restituito un intero, risultato del prodotto
     scalare

 *** Indicatori di errore ***
 Se il valore di ritorno e' EXIT_SUCCESS la routine e'
     terminata con successo

 *** Routine ausiliarie richiamate ***
 Nessuna

 *** Tempo di esecuzione ***
L'algoritmo implementato esegue 2N operazioni (1 somma ed
    una moltiplicazione)
floating-point
 *** Memoria richiesta ***
L'algoritmo implementato utilizza due array interi di
    dimensione N per la memorizzazione
di a e b, e una variabile per memorizzare il risultato.
Quindi la complessita' di spazio e' O(n).

 *** Accuratezza fornita ***
Per verificare l'accuratezza della routine sono stati
    inseriti valori numerici pari a 1,
in modo tale che il risultato sia uguale ad N.
 *** Esempio d'uso ***
Ecco come compilare il sorgente:
$gcc -o dot dot_product.c
Ecco come far girare l'eseguibile:
./dot
```

```
Alcuni casi d'uso:
- ho due vettori: a = {1.0,1.0,1.0}, b={1.0,1.0,1.0}.
    Lanciando l'eseguibile otterro'
Risultato:3
- ho due vettori: a = {2.0,1.0,1.0}, b={1.0,1.0,1.0}.
    Lanciando l'eseguibile otterro'
Risultato: 4
- ho due vettori: a = {-1.0,1.0,1.0}, b={1.0,1.0,1.0}.
    Lanciando l'eseguibile otterro'
Risultato: 2
*/


int dot_product(double *a, double *b, size_t n);

// ROUTINE CHIAMANTE
int
main(int argc, char* argv[])
{
        double *a,*b;
        int size,i;
        struct timeval start,finish;
        double duration;

        //Controlla che il numero di parametri siano
            corretti, altrimenti ritorna un errore
        if(argc<2){
             printf("Input Error\n");
             return 1;
        }

        //Legge il primo parametro passo in input che
            rappresenta la dimensione di a e b
        size = atoi(argv[1]);

        //Allocazione di a e b di size
        a = (double *) malloc (size*sizeof(double));
        b = (double *) malloc (size*sizeof(double));


        //Riempe gli array a e b con elementi 1
        for (i=0; i<size; ++i){
           a[i] = 1.0;
           b[i] = 1.0;
        }

        gettimeofday(&start, NULL);
        //stampa a video il risultato richiamando la routine
             dot_product
```

```
        //printf("Risultato:%d\n", dot_product(a, b, sizeof(
            a) / sizeof(a[0])));
        dot_product(a, b, size);
        //printf("Risultato:%d\n", );
        gettimeofday(&finish, NULL);
        duration = ((double)(finish.tv_sec-start.tv_sec)
            *1000000 + (double)(finish.tv_usec-start.tv_usec
            )) / 1000000;
        double GFLOPs = size;
        GFLOPs = GFLOPs/duration*1.0e-9;
        printf("n=%d,durata=%lf,GFLOPs=%lf\n",size,duration,
            GFLOPs);
        //libera i puntatori agli array a e b alla fine dell
            'esecuzione
        free(a);
        free(b);

        return EXIT_SUCCESS;
}

// DOT_PRODUCT: Routine che calcola il prodotto scalare
int
dot_product(double *a, double *b, size_t n)
{
        double sum = 0;
        size_t i;

        //calcola il prodotto scalare degli array a e b
            passati in input
        for (i = 0; i < n; i++) {
                sum += a[i] * b[i];
        }

        return sum;
}
```

A.1 Algoritmo di eliminazione di Gauss

- Scopo: la funzione esegue la risoluzione di un sistema di equazioni lineari del tipo $Ax = b$ utilizzando il metodo di eliminazione di Gauss con pivoting parziale.
- Descrizione: L'algoritmo esegue la fattorizzazione della matrice dei coefficienti, trasformando la matrice iniziale in una matrice triangolare superiore equivalente, da cui vengono ricavate le soluzioni del sistema, applicando il metodo della back substitution.
- Lista dei parametri (input):

  - n: dimensione del sistema
  - A: matrice dei coefficienti del sistema lineare di dimensione $n \times n$.
  - B: vettore dei termini noti di dimensione n.

- Lista dei parametri (output):

  - A: matrice dei coefficienti modificata.
  - B: vettore dei termini noti modificati.
  - x: vettore contenente le soluzioni finali del sistema.

- Indicatori di errore:

  - Il sistema potrebbe terminare restituendo errore = -1 se la matrice A è singolare

- Funzioni ausiliarie:

  - Per la misurazione del tempo di esecuzione è stata utilizzato un wrapper della funzione *gettimeofday* di C per ottenere la misura in microsecondi.

- Complessità computazionale: la complessità totale dell'algoritmo è

$$T(n) = \frac{2}{3}n^3$$

  .

- Accuratezza fornita: poiché il metodo di gauss appartiene alla categoria dei metodi risolutivi diretti, la soluzione ottenuta risulta esatta a meno di un errore di round off legato alla precisione della macchina.

Sorgente

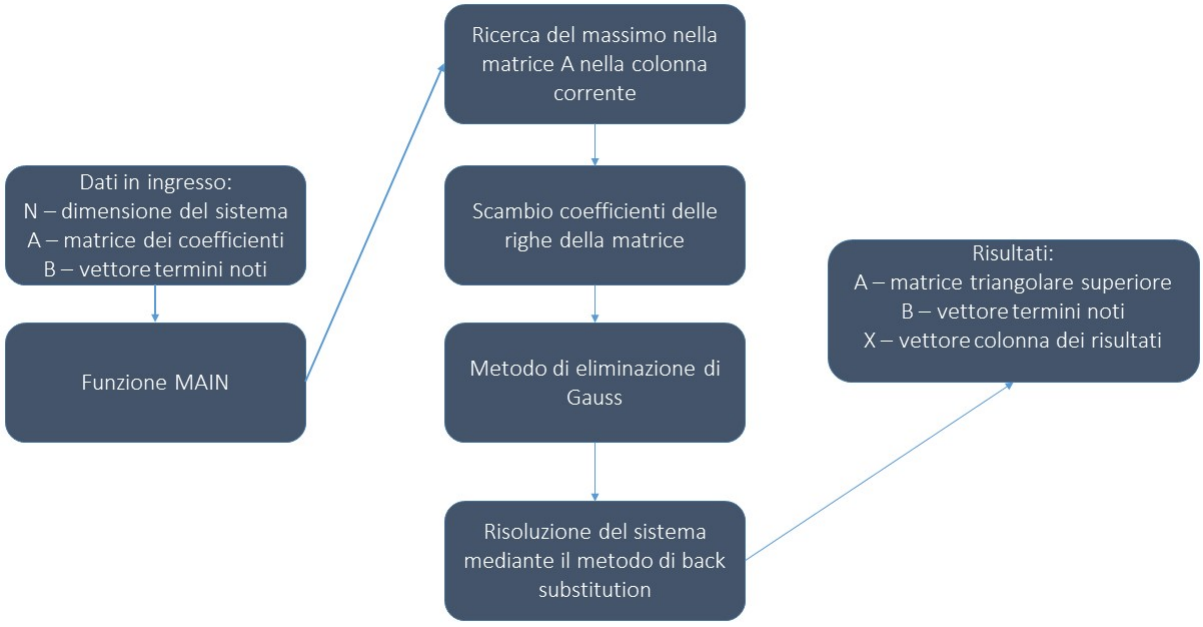
Di seguito l'algoritmo in pseudocodice:

```
procedure Gauss(in: A,b,n; out: A,b)
var: n : integer
var: a(n,n) : real
var: b(n) : real
var: i,j,k : integer

for k = 1 to n-1 {passi algoritmo}
        for i = k + 1 to n
                c = a(i,k) / a(k,k) {calcolo moltiplicatore}
```

```
                for j = k +1 to n
                        a(i,j) = a(i,j) - c × a(k,j) {
                            operazioni sulla matrice attiva}
                endfor
                b(i) = b(i) - c × b(k) {operazioni sul
                    vettore attivo}
                a(i,k) = 0 {annullamento elemento i-esima
                    colonna}
        endfor
endfor
end procedure Gauss
```

Per meglio comprendere l'organizzazione del programma viene fornito di seguito uno schema che riassume le fasi del programma che risolve un sistema di equazioni lineari $Ax = b$ con il metodo di eliminazione di Gauss.




```
        program gauss_pivoting
        !SCOPO: Risolve un sistema di equazioni lineari
            mediante
        !               la trasformazione del sistema in uno
             triangolare superiore
        !               e applicando la back substitution
        !INPUT: a matrice di reali
```

```
!               b vettore di reali dei termini noti
!               n dimensione della matrice
!OUTPUT: a matrice triangolare superiore
!                b vettore dei termini noti
implicit none
real, dimension (:,:), allocatable :: a
real, dimension (:), allocatable :: b, x
double precision, dimension (:), allocatable ::
    timeloc
integer :: i, j, k, index_max, index_reps
integer :: n, auto, reps
real*8 :: c, sumtot, max, store
integer,parameter :: seed = 86456 !seed per il
    generatore di numeri random
double precision btime, etime, time_tot

reps = 10 !l'algoritmo viene ripetuto 10 volte per
    mediare i tempi
!leggi la dimensione della matrice a
print*, "inserisci la dimensione della matrice A"
read*, n
! alloca memoria per A b e x
allocate ( a(n,n) )
allocate ( b(n) )
allocate ( x(n) )
allocate ( timeloc(reps))
!inizializzazione del generatore di numeri random
call srand(seed)
!nel caso in cui l'utente inserisca 1 la matrice
    viene generata automaticamente
print*, "Vuoi generare automaticamente la matrice
    con i dati?"
read*, auto
if(auto == 1) then
        do, i=1,n
                b(i) = 1.0
                do, j=1,n
                        a(i,j) = rand()
                enddo
        enddo
else
        !l'utente inserisce manualmente i dati
        print*,"Inserisci la matrice a"
        read(*,*) a
        print*,"Inserisci il vettore termini noti b"
        read(*,*) b
endif
do, index_reps=1,reps
        call get_cur_time(btime) ! inizio
```

```
                    misurazione tempo
C   PIVOTING
                do, k=1,n-1 !passi dell'eliminazione di
                    gauss
                  max = abs(a(k,k))
                  index_max = k
                  do, i=k+1,n !cerca il massimo per il
                      pivoting
                    IF (abs(a(i,k)) > max) THEN
                      max = abs(a(i,k))
                      index_max = i
                    endif
                  enddo
                ! Controlla che la matrice non sia
                    triangolare
                if(max == 0.0) then
                        write(*,*) "La matrice e' singolare"
                        call EXIT(-1)
                end if
                if (index_max /= k) then !scambio effettivo
                    delle righe
                        do j=k,n
                        store = a(k,j)
                        a(k,j) = a(index_max,j)
                        a(index_max,j) = store
                        end do
                store = b(k) !scambio termini noti
                b(k) = b(index_max)
                b(index_max) = store
                end if
C   ELIMINAZIONE DI GAUSS
                  do, i=k+1,n
                    c = a(i,k)/a(k,k) !calcola il
                        moltiplicatore in c
                    do, j=k+1,n !cicla sulle colonne della
                        matrice
                        a(i,j) = a(i,j) - c * a(k,j) !
                            trasformazione della matrice
                            attiva
                    enddo
                    a(i,k)=0 !cancella l'elemento in
                        posizione i,k
                              !(per ottenere la L basta
                                  assegnare c ad a(i,k))
                    b(i) = b(i) - c * b(k) !calcola l'i-
                        esimo elemento in b
                  enddo
                enddo
C   MATRICE TRIANGOLARE
```

```
                print*,"La matrice A:"
                do, i=1,n
                    write(*,*) ( a(i,j), j=1,n )
                enddo
                print*,"Il vettore b:"
                do, i=1,n
                    write(*,*) b(i)
                enddo
C   PASSO DI BACK SUBSTITUTION
                x(n) = b(n) / A(n,n) !calcola l'ultima
                    incognita
                do, i = n-1, 1, -1  !procedi all'indietro
                   sumtot = 0.0
                   do, j = i+1, n
                        sumtot = sumtot + a(i,j) * x(j)
                   enddo
                    x(i) = (b(i) - sumtot) / a(i,i)
                enddo
C   SOLUZIONE DEL SISTEMA
                print*,"La soluzione del sistema:"
                do, i=1,n
                    write(*,*) x(i)
                enddo
                call get_cur_time(etime) !fine del blocco da
                     misurare
        timeloc(index_reps) = etime - btime
        !print*,'elaps:',etime-btime
        print*,'timeloc:',timeloc(index_reps)
        enddo

        print*, 'n:', n, 'tempo impiegato:', sum(timeloc)/10
        deallocate (a) !deallocazione della memoria
        deallocate (b)
        deallocate (x)
        deallocate (timeloc)
        end program gauss_pivoting
```

Risultati

L'architettura utilizzata è la seguente: Intel® Core^{TM} i5-3470S CPU @ 2.90GHz. I risultati ottenuti sono stati condotti a partire da $n = 5$ a n = 2000 (vedi tabella A.6), è possibile ottenere una stima dei tempi anche per $n > 2000$ calcolando il fattore di crescita all'aumentare della dimensione del sistema. Come è possibile osservare dalla tabella A.7 il tempo di esecuzione cresce con un fattore di circa 7 per cui si ottiene che con dimensioni di $n = 16000$ si otterrebbero tempi di esecuzione di circa $T(N) = 9500$ *secondi*.

Processor Number	*Frequency Type*	*Clock (GHz)*	P_{peak}
i5-3470S	Base	2.9	92.8
	Singole Core	3.6	115
	GPU Only	1.1	52.8

Tabella A.5: Architettura utilizzata per i test

Un ulteriore confronto effettuato è quello con le routine contenute nella

n	*T(N)*
10	0.4196E-05
20	0.2611E-04
100	0.2855E-02
200	0.1752E-01
1000	0.2254E+01
2000	0.1841E+02

Tabella A.6: Tempi di esecuzione per l'algoritmo di esecuzione di Gauss

Fattore	*2*n/n*
20/10	6.22
200/100	5.95
2000/1000	8.16

Tabella A.7: Fattori di crescita dei tempi di esecuzione dell'algoritmo di Gauss

libreria matematica **LAPACK** che sfrutta le routine base per l'algebra lineare contenute in **BLAS**, è possibile osservare che con input piccoli (fino a $n = 100$) l'algoritmo di Gauss ha tempi minori rispetto a quello implementato nella routine di LAPACK mentre al crescere di n si ottengono tempi migliori sfruttando la routine sgesv della libreria (che a sua volta usa sgetrf e sgetrs), per maggiori approfondimenti (http://www.netlib.org/lapack/).

	Gauss	*Gauss LAPACK*
10	0.4196E-05	0.1400E-03
20	0.2611E-04	0.1159E-03
100	0.2855E-02	0.4010E-03
200	0.1752E-01	0.8111E-03
1000	0.2254E+01	0.3986E-01
2000	0.1841E+02	0.2889E+00

Tabella A.8: Confronto dell'implementazione con la routine della libreria di LAPACK ***sgesv***

Appendice B

Configurazione dei test case del software ROMS, WC12 e WC13

Per la configurazione del test case WC13 ovvero quello relativo al CCS è sufficiente seguire il tutorial riportato sul wiki di ROMS (fare riferimento a [46]). In generale ROMS ha bisogno dei seguenti file per l'esecuzione:

- Grid File: rappresenta la griglia del dominio da andare ad analizzare
- Initial: è il file contenente le condizioni iniziali
- Forcing: sono i file...
- Boundary: sono i file dei confini
- Observation: contiene i dati sulle osservazioni in un arco temporale raccolte dai sensori geograficamente distribuiti.

In particolare per il test case WC13 c'è bisogno dei seguenti file:

```
Grid File:   ../Data/wc13_grd.nc
Nonlinear Initial File:   wc13_ini.nc
Forcing File 01:   ../Data/coamps_wc13_lwrad_down.nc
Forcing File 02:   ../Data/coamps_wc13_Pair.nc
Forcing File 03:   ../Data/coamps_wc13_Qair.nc
Forcing File 04:   ../Data/coamps_wc13_rain.nc
Forcing File 05:   ../Data/coamps_wc13_swrad.nc
Forcing File 06:   ../Data/coamps_wc13_Tair.nc
Forcing File 07:   ../Data/coamps_wc13_wind.nc
Boundary File:   ../Data/wc13_ecco_bry.nc
```

```
Initial Conditions STD File:   ../Data/wc13_std_i.nc
Boundary Conditions STD File:   ../Data/wc13_std_b.nc
Surface Forcing STD File:   ../Data/wc13_std_f.nc
Initial Conditions Norm File:   ../Data/wc13_nrm_i.nc
Boundary Conditions Norm File:   ../Data/wc13_nrm_b.nc
Surface Forcing Norm File:   ../Data/wc13_nrm_f.nc
Observations File:   wc13_obs.nc
```

per facilitare il compito all'utente che si avvicina a ROMS per la prima volta, nella directory del test si trova uno script *job_i4dvar.sh* che si occupa di predisporre questi file per una corretta esecuzione del modulo I4DVAR di ROMS. Gli altri due file fondamentali per far funziona in maniera opportuna ROMS sono *ocean_wc13.in* e *s4dvar.in* il primo file di input in cui sono contenute tutte le possibili configurazioni con cui si può far partire ROMS ed in particolare quella che ci interessa per questo lavoro ovvero *Ntilei* ed *Ntilej* ovvero l'opzione relativa al partizionamento del dominio quando si coinvolgono più processori o più thread nell'esecuzione. Il secondo file *.in* è relativo alle opzioni relative al modulo I4DVAR.
Per lanciare ROMS (nel nostro caso compilato con la libreria MPI) ci basterà utilizzare questo comando:

```
mpirun -np 4 oceanM ocean_wc13.in > & log &
```

dove:

- -np rappresenta il numero di processori coinvolti nell'esecuzione
- oceanM è il nome dell'eseguibile
- ocean_wc13.in è il file di input descritto precedentemente
- >& serve a redirigere l'ouput su un file chiamato log

Alla fine dell'esecuzione in output tramite gli opportuni script matlab sarà possibile plottare i risultati, ad esempio in una esecuzione di test il risultato della funzione test ottenuto con ROMS potrebbe avere la forma in Figura B.1.

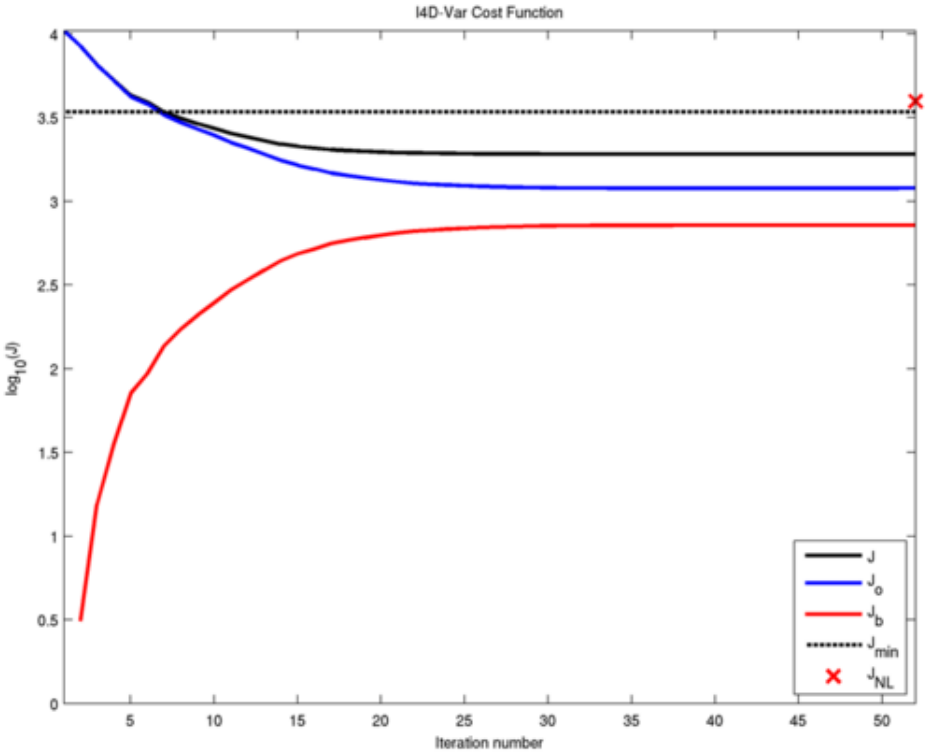


Figura B.1: Plot dell'ouput della funzione costo per il test case WC13 [Fonte - ROMS Tutorial IS4DVAR]

Appendice C

Installazione di ROMS su una architettura HPC: Il Mont-Blanc al BSC

Per l'installazione del software ROMS sul cluster del BSC si è reso necessaria l'installazione delle librerie netcdf poichè allo stato attuale il cluster non ne è fornito di default.

C.0.1 Installazione di NetCDF

L'installazione di ROMS richiede in questo ordine i seguenti passi:

- Installazione della libreria HDF5 (5-1.8.18)
- Installazione della libreria NetCDF (4.4.1.1)
- Installazione della libreria NetCDF-Fortran (4.4.4)

le variabili d'ambiente da impostare sono le seguenti:

```
FC Fortran Compiler
CC C Compiler
NETCDF NetCDF installation path
LDFLAGS Library directory location
CPPFLAGS Include directory location
```

dopodichè è possibile procedere con l'installazione di ROMS. La libreria HDF5 è stata configurata come segue:

```
$./configure --prefix=/home/ldamore/opt --enable-static-exec
    \
  --disable-shared 2>&1 | tee conf.log
```

l'opzione *–prefix* è necessaria per indicare allo script di configurazione dove installare la libreria, dopodichè è possibile effettuare la configurazione di netcdf con questo comando:

```
$./configure CPPFLAGS=-I/home/ldamore/opt/include \
LDFLAGS=-L/home/ldamore/opt/lib LIBS="-ldl -lm" \
--prefix=/home/ldamore/opt --disable-shared \
2>&1 | tee conf.log
```

e della version fortran di netcdf

```
./configure CPPFLAGS=-I/home/ldamore/opt/include \
LDFLAGS=-L/home/ldamore/opt/lib \
LIBS="-lnetcdf -lhdf5_hl -lhdf5 -lz -ldl -lm -lcurl" \
--prefix=/home/ldamore/opt --disable-shared \
2>&1 | tee conf.log
```

ciascuna configurazione deve quindi essere seguita dalla coppia di comandi *make* e *make install* (è consigliabile eseguire sempre un *make test* che richiede un lungo tempo di completamento ma è utile a sapere se l'installazione è andata a buon fine).

C.0.2 Installazione di ROMS

Il comando per scaricare dal repository l'ultima versione di ROMS ha questa forma:

```
$svn checkout --username joe_roms https://www.myroms.org/svn
    /src/trunk MyDir
```

in questo modo avremo scaricare il codice sorgente di ROMS, e successivamente si può scaricare la directory relativa ai test:

```
$svn checkout --username joe_roms https://www.myroms.org/svn
    /src/test MyDir
```

Il file da modificare per l'installazione di ROMS ha questa forma *SO-Compilatore.mk* dove SO rappresenta il sistema operativo utilizzato, e compilatore si riferisce al compilatore Fortran installato sul sistema. Nel caso del Mont-Blanc è stato utilizzato il file *Linux-gfortran.mk*. Le variabili da modificare all'interno dello script sono:

```
export NF_CONFIG=/home/ldamore/opt/bin/nf-config
export NETCDF_INCDIR=/home/ldamore/opt/include
```

che si riferiscono al percorso di installazione della libreria NETCDF. A questo punto è possibile entrare nella directory di ROMS relativa al test case che interessa mandare in esecuzione, in questo lavoro sono stati utilizzati i test case WC12 e WC13.

Bibliografia

[1] Jack Dongarra, Ian Foster, Geoffrey Fox, William Gropp, Ken Kennedy, Torczon Linda, and Andy White. *The Sourcebook of Parallel Computing*. 2003.

[2] Susan L. Graham, Peter B. Kessler, and Marshall K. Mckusick. gprof: a Call Graph Execution Profiler. *ACM SIGPLAN Notices*, 17(6):120–126, 1982.

[3] J.W. Demmel et al. Note dal corso di Applications of Parallel Computers, Berkeley University, 2011.

[4] Almerico Murli. *Lezioni di Calcolo Parallelo*. 2004.

[5] L D Fosdick. *An Introduction to High-performance Scientific Computing*. Scientific and engineering computation. MIT Press, 1996.

[6] Venkatramani Balaji, Eric Maisonnave, Niki Zadeh, Bryan N. Lawrence, Joachim Biercamp, Uwe Fladrich, Giovanni Aloisio, Rusty Benson, Arnaud Caubel, Jeffrey Durachta, Marie Alice Foujols, Grenville Lister, Silvia Mocavero, Seth Underwood, and Garrett Wright. CPMIP: Measurements of real computational performance of Earth system models in CMIP6. *Geoscientific Model Development*, 10(1):19–34, 2017.

[7] J. J. Dongarra. Performance of various computers using standard linear equations software in a Fortran environment. *SIGARCH Comput. Archit. News*, 20(3):22–44, 1992.

[8] Almerico Murli. *Lezioni di laboratorio di programmazione e calcolo*. 1999.

[9] Samuel Williams, Andrew Waterman, and David Patterson. Roofline: an insightful visual performance model for multicore architectures. *Communications of the ACM*, 52(4):65–76, 2009.

[10] Gene M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the April 18-20, 1967, spring joint computer conference on - AFIPS '67 (Spring)*, page 483, 1967.

[11] Ananth Grama, Anshul Gupta, George Karypis, and Vipin Kumar. Introduction to Parallel Computing; 2nd Edition. *Search*, page 856, 2003.

[12] David Henty. Lectures of "Introduction to OpenMP and MPI" - Scalability. https://www.archer.ac.uk/training/course-material/2015/12/OpenMPandMPI_Portsmouth/index.php.

[13] Intel. *Intel® 64 and IA-32 Architectures Software Developer's Manual*. 2016.

[14] "Netlib Repository". Benchmark Programs and Reports, 2014.

[15] Jack J. Dongarra, Piotr Luszczek, and Antoine Petitet. The LINPACK Benchmark: past, present and future. *Concurrency and Computation: Practice and Experience*, 15(9):803–820, aug 2003.

[16] Gene H. Golub Charles F. Van Loan. *Matrix Computation*, volume 53. 2013.

[17] Max Planitz. LAPACK Users Guide. *The Mathematical Gazette*, 79(484):210, 1995.

[18] Blas. Basic Linear Algebra Subprograms (BLAS) FAQ. *Netlib Repository*, page 1, 2005.

[19] Jack Dongarra. HPL Documentation. http://www.netlib.org/benchmark/hpl/documentation.html.

[20] Jack J. Dongarra. HPL - Algorithm. www.netlib.org/benchmark/hpl/algorithm.html.

[21] Jack Dongarra and Michael A Heroux. Toward a new metric for ranking high performance computing systems. *Sandia Report, SAND2013-4744*, 312(June):1–19, 2013.

[22] Jack Dongarra, Piotr Luszczek, and Michael Heroux. HPCG Benchmark. http://www.hpcg-benchmark.org/, 2017.

[23] Yuuichirou Ajima, Tomohiro Inoue, Shinya Hiramoto, and Toshiyuki Shimizu. Tofu: Interconnect for the K computer. *Fujitsu Scientific and Technical Journal*, 48(3):280–285, 2012.

[24] Robert Alverson, Duncan Roweth, and Larry Kaplan. The Gemini System Interconnect. In *2010 18th IEEE Symposium on High Performance Interconnects*, pages 83–87. IEEE, aug 2010.

[25] J McCalpin. STREAM: Sustainable Memory Bandwidth in High-Performance Computers. *Silicon Graphics Inc*, 1995.

[26] John D McCalpin. Memory Bandwidth and Machine Balance in Current High Performance Computers. *IEEE Computer Society Technical Committee on Computer Architecture (TCCA) Newsletter*, (May):19–25, 1995.

[27] S. Browne. A Portable Programming Interface for Performance Evaluation on Modern Processors. *International Journal of High Performance Computing Applications*, 14(3):189–204, 2000.

[28] J Dongarra, K London, and S Moore. Using PAPI for hardware performance monitoring on Linux systems. *Proc. Conf. on Linux ...*, pages 25–27, 2001.

[29] Smeds Nils. PAPI predefined named events. http://icl.cs.utk.edu/projects/papi/files/html_man/papi_presets.html.

[30] Samuel Williams, Andrew Waterman, and David Patterson. Roofline. *Communications of the ACM*, 52(4):65, 2009.

[31] Samuel Webb Williams. Auto-tuning Performance on Multicore Computers. *Phd Thesis: Berkeley*, page 241, 2008.

[32] Samuel Williams, Brian Van Straalen, Terry Ligocki, Leonid Oliker, Matthew Cordery, and Linda Lo. Roofline Performance Model. https://crd.lbl.gov/departments/computer-science/PAR/research/roofline/.

[33] Anwar Ali. Intel 64 and IA-32 Architectures Optimization Reference Manual. *Intel Technology Journal*, 09(03):1–660, 2005.

[34] Rossella Arcucci, Luisa Carracciuolo, and Almerico Murli. A Scalable Approach for Variational Data Assimilation. pages 239–257, 2014.

[35] R Arcucci, Simone Celestino, G Laccetti, and A Murli. A Scalable Numerical Algorithm for solving Tikhonov Regularization Problems in a large scale application.

[36] Andrew M. Moore, Hernan G. Arango, Gregoire Broquet, Brian S. Powell, Anthony T. Weaver, and Javier Zavala-Garay. The Regional Ocean Modeling System (ROMS) 4-dimensional variational data assimilation systems. Part I - System overview and formulation. *Progress in Oceanography*, 91(1):34–49, 2011.

[37] Hernan G. Arango. ROMS Website. https://www.myroms.org/.

[38] Hernan G. Arango. ROMS Documentation - Nested Grids. https://www.myroms.org/wiki/Nested_Grids.

[39] R. B. Lehoucq, D. C. Sorensen, and C. Yang. ARPACK Users' Guide. *Communication*, 6:147, 1998.

[40] S. Balay, WD Gropp, LC McInnes, and BF Smith. Efficient Management of Parallelism in Object Oriented Numerical Software Libraries, Modern Software Tools in Scientific Computing, Ed. E. Arge, AM Bruaset and HP Langtangen, 163-202. *Modern Software Tools in Scientific Computing*, pages 163–202, 1997.

[41] Italo Epicoco, Silvia Mocavero, Francesca Macchia, and Giovanni Aloisio. The roofline model for oceanic climate applications. In *Proceedings of the 2014 International Conference on High Performance Computing and Simulation, HPCS 2014*, pages 732–737, 2014.

[42] Andrew M. Moore, Hernan G. Arango, Gregoire Broquet, Chris Edwards, Milena Veneziani, Brian Powell, Dave Foley, James D. Doyle, Dan Costa, and Patrick Robinson. The Regional Ocean Modeling System (ROMS) 4-dimensional variational data assimilation systems. Part III - Observation impact and observation sensitivity in the California Current System. *Progress in Oceanography*, 91(1):74–94, 2011.

[43] ARM. Cortex^{TM} - A15 - Technical Reference Manual. Technical report, 2011.

[44] José Fonseca. gprof2dot. https://github.com/jrfonseca/gprof2dot.

[45] William Von Hagen. The Definitive Guide To GCC. *The Definitive Guide To GCC*, page 550, 2006.

[46] Arango and Robertson. Incremental, Strong Constraint, 4D-Var (I4D-Var) Data Assimilation.

Elenco delle figure

Elenco delle tabelle