Relazione Percorso d'Eccellenza – Ottimizzazione NUMA

Autore Emanuele De Santis

Supervisore Prof. Francesco Quaglia

Il Percorso d'Eccellenza ha riguardato lo studio dell'architettura NUMA (Non Uniform Memory Access) e l'ottimizzazione dei processi che vengono eseguiti su tale architettura.

Durante la prima fase del lavoro, una volta individuato l'argomento, si è iniziato a studiare il funzionamento di base di sistemi di calcolo NUMA, sia attraverso materiale prodotto da Red Hat[1] che attraverso una pubblicazione del DIAG[2].

L'architettura NUMA è una architettura di memoria per sistemi multiprocessore dove la memoria e le CPU sono partizionate in blocchi detti **nodi**. Ogni nodo contiene almeno una CPU e una parte della memoria principale (non obbligatoriamente uguale per ogni nodo). Il tempo di accesso a una partizione della memoria dipende strettamente da quale CPU esegue l'accesso: sarà minore se la CPU appartiene allo stesso nodo della partizione della memoria, altrimenti sarà maggiore. Le latenze vengono introdotte da controller hardware che servono sia per il passaggio dei dati che per l'arbitraggio nel caso di accessi concorrenti.

```
node distances:
node   0   1   2   3   4   5   6   7
  0:  10  20  20  20  20  20  20  20
  1:  20  10  20  20  20  20  20  20
  2:  20  20  10  20  20  20  20  20
  3:  20  20  20  10  20  20  20  20
  4:  20  20  20  20  10  20  20  20
  5:  20  20  20  20  20  10  20  20
  6:  20  20  20  20  20  20  10  20
  7:  20  20  20  20  20  20  20  10
```
[3]

Questa architettura, se ben gestita, permette un miglioramento delle performance generali poiché al livello del protocollo firmware che governa gli accessi alla memoria, ogni CPU deve sincronizzarsi con (ed eventualmente attendere) solo un sottoinsieme delle CPU installate per poter accedere alla memoria (cioè deve eventualmente attendere solo quelle che sono attestate sullo stesso nodo), mentre in un'architettura tradizionale (basata sul paradigma UMA – Uniform Memory Access) la coda di accesso alla memoria sarebbe formata potenzialmente da tutte le CPU.

---

[1] Rik van Riel, Vinod Chegu - "Automatic NUMA Balancing", Red Hat Summit 2014

[2] Ilaria Di Gennaro, Alessandro Pellegrini, Francesco Quaglia – "OS-based NUMA Optimization: Tackling the Case of Truly Multi-Thread Applications with Non-Partitioned Virtual Page Accesses"

[3] Ottenuto tramite comando bash "numactl -H" che mostra la configurazione hardware NUMA presente

Si è passati successivamente alla fase di ricerca di un tool che permettesse di ottimizzare automaticamente ed a runtime l'esecuzione di processi su architettura NUMA. Il tool di base presente su molte distribuzioni Unix è ***numactl***, che permette una configurazione manuale e per processo dello scheduling NUMA e della policy di memory placement delle pagine logiche e del processo stesso, modificando l'affinità di processore per i thread attivi nel processo o imponendo che la memoria allocata appartenga ad un determinato nodo dell'architettura NUMA[4]. C'era l'esigenza, però, di identificare e sperimentare un sistema che gestisse anche automaticamente queste politiche per tutti i processi e che attuasse le necessarie contromisure in caso di accessi non ottimizzati a memoria (ad esempio eseguiti da un thread attivo sulla CPU0 presente nel nodo 0 verso la memoria fisica nel nodo 1).

Il tool che si è utilizzato è stato quindi il demone ***numad***, che, mandato in avvio secondo schemi di default, scansiona tutti i processi (ogni 15 secondi) alla ricerca di possibili ottimizzazioni. In particolare, il demone cerca di spostare thread o pagine di memoria affinché i tempi di accesso a queste ultime siano minori possibili. Questo tool non gestisce processi che non rispettino una soglia significativa (che è attualmente fissata a 300MB di memoria allocata e l'utilizzo di almeno metà, ovvero il 50%, di una CPU).[5]

La sperimentazione basata sull'utilizzo del demone è stata orientata a valutare l'efficacia di tale demone per il caso di applicazioni altamente parallele basate su thread concorrenti associati allo stesso spazio di indirizzamento. Tale scenario e' stato identificato come difficile da trattare in letteratura, per cui solo un numero ridotto di approcci hanno dimostrato la loro efficacia. D'altro canto non sembrano esistere sperimentazioni con tali tipi di applicazioni tese a valutare ***numad***. Infatti, nella letteratura disponibile non si fa riferimento al campo d'azione di ***numad***, e quindi non si conosce a priori l'ottimizzazione resa a partire da generici contesti applicativi (ad esempio se funziona meglio in scenari multiprocess oppure multithread).

Per sperimentare l'efficacia del demone, questo è stato fatto girare su una macchina x86_64 con 32 CPU e 64GB di RAM, con 8 nodi NUMA da 4 CPU e 8GB di memoria ciascuno. La versione del kernel linux utilizzata è "*2.6.32-5-amd64*" [6] ed è stato avviato numad con parametri in input "**`numad -i 20 -l 7`**". Il parametro "**-i**" indica l'intervallo di tempo (in secondi) che numad attende fra ogni scan del sistema ed in questo caso è stato impostato a 20 secondi. Il parametro "**-l**" invece indica il livello di log (che di default è 5 e il massimo livello impostabile è 7). Il log che numad utilizza si trova in */var/log/numad.log* .

Si è progettato, quindi, uno scenario di test che presenta un gran numero di thread (uno per ogni CPU installata sul sistema) e che consente due pattern differenti di accesso a memoria. Questo

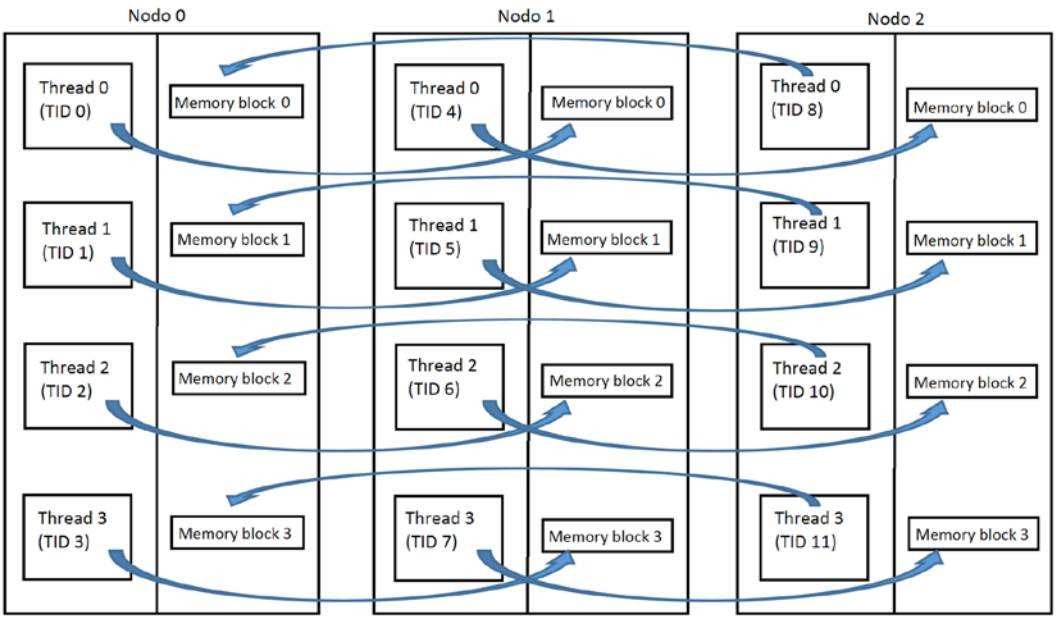
[4] Da: Linux man page - numactl
[5] Da: Linux man page - numad
[6] Ottenuto tramite il comando bash "uname -a"

scenario ha l'obiettivo di verificare il comportamento di numad in caso di forte concorrenza su blocchi di memoria nello stesso address space da parte di thread concorrenti.

La struttura di base, comune ad entrambi gli schemi di accesso, è così composta:

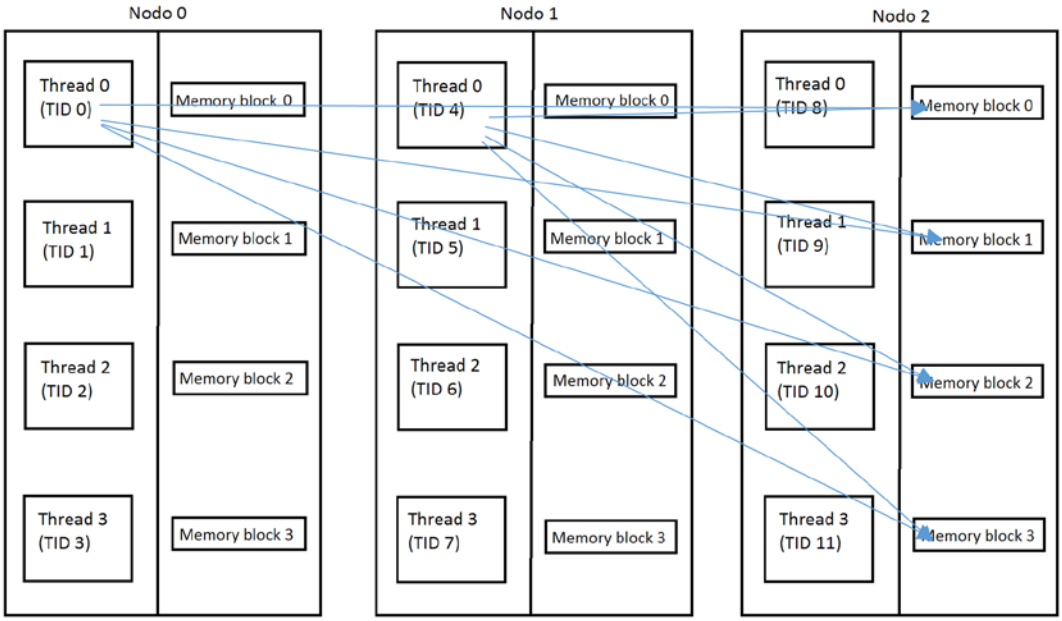
- È presente numero di thread pari al numero di CPU installate sul sistema
- È presente numero di blocchi di memoria pari al numero dei thread
- Su ogni nodo sono presenti un numero di thread e un numero di blocchi di memoria pari al numero di CPU installate su quel nodo
- L'affinità del thread i-esimo è impostata inizialmente sulla CPU i-esima; successivamente viene settatta a `0xffffffff` (cioè viene fatta scegliere al sistema operativo, come di default)

Il primo pattern di accesso prevede che l'i-esimo thread in un nodo acceda all'i-esimo blocco di memoria su un altro nodo.



Inoltre, tramite un parametro, è possibile associare l'i-esimo thread non all'i-esimo blocco di memoria ma a un blocco scelto casualmente fra quelli del nodo successivo.

Il secondo pattern di accesso invece prevede che thread in due nodi consecutivi accedano a tutta la memoria allocata su un terzo nodo

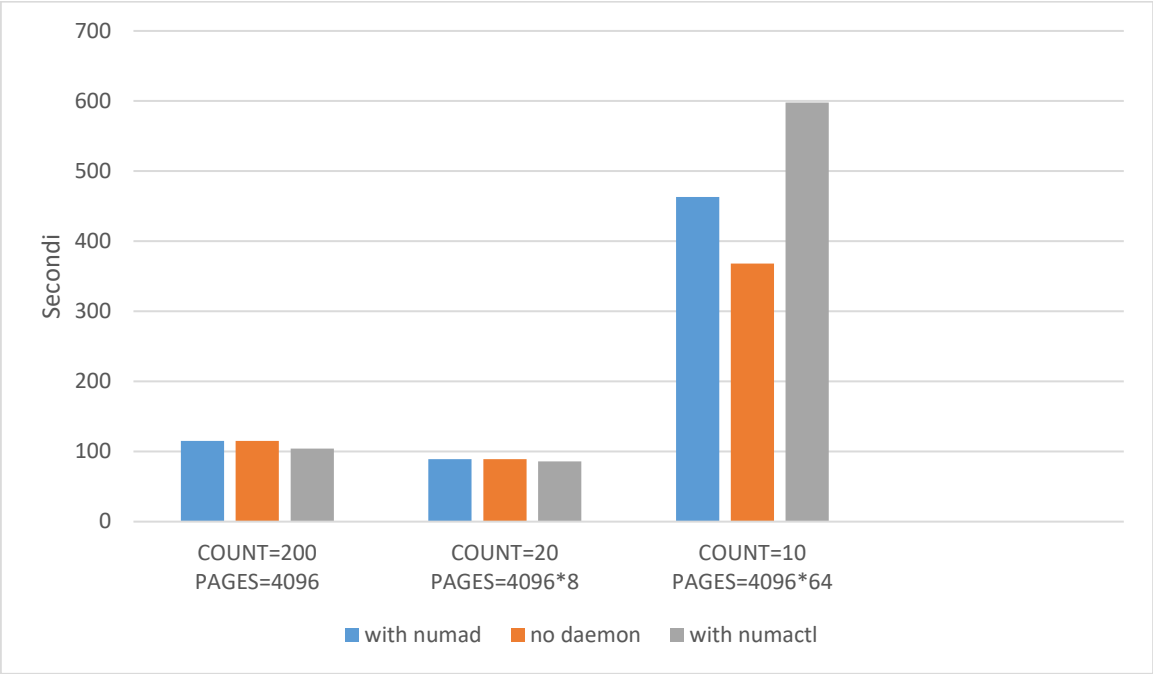


Il main thread, in base ad un parametro (configurabile in compilazione) sceglie se utilizzare il primo o il secondo schema di accesso, dopodiché si sincronizza sulla terminazione di tutti i thread che ha creato. Inoltre, al momento dell'allocazione dei vari blocchi, scrive un byte su ogni pagina di memoria in modo da prevenire che essa venga mappata in RAM sul frame empty-zero (non contribuendo quindi realmente all'uso di memoria fisica).

Il cambio di affinità viene gestito da un altro processo, che riceve i Thread ID (TID) dei vari thread, come definiti al livello kernel (non al livello della libreria `pthread`), tramite il comando shell **`ps -eLf | grep a.out`** con redirect dello standard output su un file. Questo comando lista tutti i thread con i relativi PID e TID correntemente attestati sul sistema (e in questo caso si prendono solo quelli di interesse, cioè quelli relativi all'applicazione *a.out*, ovvero al benchmark descritto prima). Una volta collezionati tutti i TID tramite lettura del file generato, viene impostata l'affinità con la bitmask `CPU_set=1<<thread_id` tramite la system call `sched_setaffinity()` e dopo un tempo parametrico (impostato a 3 secondi) si ripassa alla maschera di affinità di default (`0xffffffff`).

Questo micro-benchmark così sviluppato dovrebbe evidenziare carenze, qualora ci fossero, nelle ottimizzazioni svolte da numad, soprattutto poiché l'accesso alla memoria è concorrente (in entrambi i pattern) e svolto da molteplici thread contemporaneamente su pagine che possono afferire a nodi differenti dell'architettura NUMA. Tutto ciò dovrebbe rendere più difficile la scelta da parte del demone su come e in che modo spostare thread o pagine di memoria da un nodo a un altro per migliorare le prestazioni complessive del processo. In altre parole, si è cercato di creare uno stress-test basato su patter di accesso alla memoria articolato e di difficile gestione, in termini di ottimizzazione dell'affinità thread/pagine.

Dopo aver svolto una sessione di test utilizzando il secondo pattern di accesso e alcuni valori per il numero di pagine allocate e il numero di cicli di accesso su queste pagine, sono stati ottenuti i seguenti risultati:



I valori sono stati ottenuti dalla media di più run con gli stessi parametri, inoltre i valori "with numactl" sono stati ottenuti tramite il comando **`numactl --membind=1,3,5,7`**, imponendo quindi che le allocazioni di memoria avvenissero solo sui nodi 1,3,5,7.

Questi dati evidenziano come numad non dà vantaggi in termini di tempo di esecuzione del micro-benchmark, e anzi, a causa dell'overhead dovuto all'inutile spostamento di pagine di memoria, può rendere le prestazioni ancora peggiori (si nota infatti che all'aumentare del numero di pagine, in caso numad sia attivo, aumenta di molto il tempo di esecuzione del micro-benchmark).

In conclusione, è possibile affermare che, in caso di applicazioni con grande concorrenza multithread, numad non riesce ad identificare lo schema di accesso a memoria correttamente e, per questo motivo, compie azioni di ottimizzazione errate.