IMPLEMENTAZIONE DI MPI_ALLREDUCE CON QUANTIZZAZIONE

Luca Mautino

Matricola : 1997410
mautino.1997410@studenti.uniroma1.it

Saverio Pasqualoni

Matricola : 1845572
pasqualoni.1845572@studenti.uniroma1.it

ABSTRACT

Il nostro progetto ha lo scopo di testare lo speedup nell'implementazione di una collective, in particolare MPI_Allreduce, su dati quantizzati.
La quantizzazione è un processo di compressione che consiste nel rappresentare un range di valori reali (più precisamente la loro approssimazione in float o double) con una serie di valori discreti, di modo da diminuire la quantità di informazioni da inviare attraverso una rete.
Allo scopo abbiamo implementato una versione custom di MPI_Allreduce attraverso la libreria PMPI per quantizzare i dati in uscita al fine di ridurre l'utilizzo di banda in ipotetici scenari di addestramento di reti neurali su cluster.
Abbiamo implementato 2 strutture di comunicazione tra i processi: *Ring* e *Recursive Halving* e 4 algoritmi di quantizzazione: *Lloyd, Non Lineare, Uniforme, Omomorfico* (in realtà come vedremo in seguito, un quinto algoritmo di quantizzazione è presente nel codice, ma si tratta di una quantizzazione omomorfica che non richiede la ricerca di massimi e minimi poiché conosciuti a priori).
La libreria OpenMP è stata utilizzata per sfruttare, dove possibile, lo speedup dato dal multithreading.
Il codice è stato testato su un cluster computazionale dal professor De Sensi e i risultati ottenuti sono stati rapportati con una baseline data da una chiamata standard di MPI_Allreduce.

1. IMPOSTAZIONE DEL LAVORO

Sin dal principio era chiaro che la più grande difficoltà del progetto sarebbe stata quella di gestire l'eterogeneità dei tipi di dato richiesti da ogni tipo di quantizzazione tramite una chiamata unica a MPI_Allreduce.
In particolare il dover utilizzare void pointers di struct e fare dei cast runtime al tipo giusto di struct (andando in un certo senso a simulare il lavoro che in altri linguaggi di programmazione è gestito dai generics) è stato molto problematico, ma il risultato è un codice leggibile e facilmente espandibile.
Infatti diversi algoritmi di quantizzazione, possono richiedere l'utilizzo di strutture dati diverse, ma al minimo, la struttura dati "base" utilizzata nel codice prevede tre elementi:
un array di *uint8_t* (o *uint16_t*) chiamato *vec* contenente i dati compressi e due float *min* e *max*.
Nel caso della quantizzazione non lineare vi è presente anche un intero *type* che indica il tipo di quantizzazione utilizzata e caso della quantizzazione con metodo di Lloyd-Max invece vi è presente un secondo array di float *codebook* di valori di ricostruzione.

2. STRUTTURA DEL CODICE

La file structure del programmma vede:

- file con algoritmi di quantizzazione (e rispettivi header):
  - *uniform_quantizer.c*
  - *non_linear_quantizer.c*
  - *lloyd_max_quantizer.c*
  - *homomorphic_quantizer.c*
  - *known_range_quantizer.c*
- file con algoritmi di riduzione custom (e rispettivi header):
  - *collectives.c* contenente la custom MPI_Allreduce
  - *ring_reduce.c* contenente l'algoritmo ad anello
  - *recursive_halving_reduce.c* contenente l'algoritmo ad albero
- file con funzioni generiche usate in varie parti del codice (e rispettivi header):
  - *tools.c* contenente le definizioni degli enumeratori, le struct per la quantizzazione (MinMax...), gli algoritmi usati per la generazione degli array, la misura dell'errore di ricostruzione e vari tools di debugging
  - *tools_collectives.c* contenente funzioni di allocazione, deallocazione, invio, ricezione delle struct e wrapper per la quantizzazione e dequantizzazione dei dati

Nel file *main.c* verranno generati **n** processi, ciascuno di essi, tramite la funzione *RandFloatGenerator()* genererà degli array di float di dimensione *dim* (parametro di input), con valori compresi tra un minimo e un massimo.
Su questi vettori di float verranno fatte due riduzioni: prima una normale MPI_Allreduce (chiamata tramite PMPI_Allreduce per via della libreria di profiling utilizzata) per avere dei valori di riferimento, poi, una volta avuta una baseline, una chiamata alla nostra custom Allreduce.
Algoritmo di quantizzazione, struttura di riduzione e numero di bits utilizzati nella quantizzazione vengono definiti tramite variabile ambientale.

3. RIDUZIONE CUSTOM AL MICROSCOPIO

La nostra funzione MPI_Allreduce risulta essere una funzione wrapper che, una volta lette le variabili ambientali, tramite switch statement, seleziona la corretta funzione di quantizzazione, per algoritmo di quantizzazione, per struttura di riduzione e per bits.

Utile notare come esistano tre versioni di *RecursiveHalvingSend*, una generica utilizzata per quantizzazioni non omomorfiche e due per quantizzazioni omomorfiche (una per 16 bits e una per 8 bits).
Prima di spiegare il perché della scelta risulta utile definire cos'è un Omomorfismo funzionale e in particolare perché ci è utile. Da wikipedia:
In algebra astratta, un omomorfismo è un'applicazione tra due strutture algebriche dello stesso tipo che conserva le operazioni in esse definite.
Calato nel nostro caso, vuol dire che, in una quantizzazione omomorfica l'operazione di addizione (imprescindibile nella riduzione del vettore) può essere applicata direttamente su dati quantizzati, senza bisogno di dequantizzarli. Questo ci permette quindi di lavorare con somme parziali di dati compressi, e quindi di poter quantizzare (e dequantizzare) i dati una sola volta, al contrario di farlo a ogni step di invio.
Proprio per questa motivazione ci siamo trovati a dover utilizzare delle funzioni diverse qualora fossimo nel caso *BITS=16* oppure *BITS=8*. Tale problema deriva dalla tipizzazione statica di C.
Al contrario, con quantizzazioni non omomorfe invece abbiamo avuto la possibilità di sfruttare delle funzioni wrapper (che gestiscono da sé in base alla variabile ambientale entrambe le dimensioni degli spazi di quantizzazione) che si occupano di allocazione e deallocazione di memoria, quantizzazione e dequantizzazione dei dati. Possibilità data dal fatto che le addizioni sarebbero sempre avvenute tra float a prescindere dalla tipologia di dato quantizzato. Tale situazione non avviene nel caso della quantizzazione ad anello dal momento che non abbiamo implementato quantizzazioni diverse da uniforme, omomorfica e known_range.

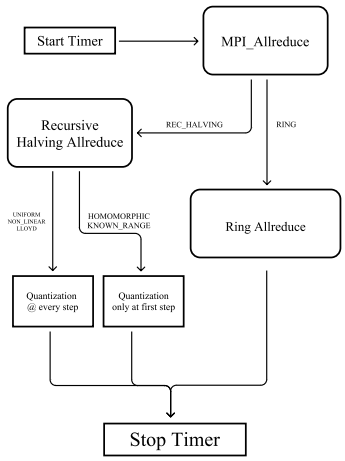

Figure 1: flowchart semplificato

4. METODI DI QUANTIZZAZIONE

Per ogni metodo di quantizzazione abbiamo implementato al minimo due funzioni di quantizzazione e due di dequantizzazione (a seconda del numero di bits scelti).
C'è però da sottolineare come molti di questi metodi (in particolare il non lineare e il Lloyd) richiedano l'utilizzo di molte funzioni ausiliari.

4.1. Uniforme

La quantizzazione uniforme consiste nell'approssimare ad un numero finito di valori di ricostruzione equidistanti tra loro il vettore di partenza.
Per permettere ciò, vengono calcolati minimo e massimo del vettore, quindi in base al numero di bits (in particolare all'intervallo di rappresentazione disponibile con quel dato numero di bits) sono definiti i valori di ricostruzione.
Ogni valore viene quindi codificato con un intero **i** rappresentante il valore di ricostruzione associato l'intervallo di quantizzazione nel quale il numero non quantizzato rientra.

4.2. Non lineare

La quantizzazione non lineare (o companding) consiste nell'applicare ai dati non quantizzati una funzione (appunto non lineare) che ha lo scopo di modificare gli intervalli effettivi di quantizzazione.
Questo ha lo scopo di mantenere più varianza per i valori vicini alla media (e quindi di conseguenza ridurla per valori agli estremi).
Viene utilizzata soprattutto nel signal processing audio, poiché si è notato che la maggior parte delle informazioni utili si trovano vicino allo zero.
Abbiamo implementato in particolare due tipi di quantizzazioni non lineari, la $\mu - law$ e la $\alpha - law$.
Da notare come entrambe, essendo pensate per il signal processing, richiedano un intervallo di dati compreso in $[-1, 1]$ e che quindi, prima dell'applicazione di una di queste due funzioni, i dati vengano schiacciati in tale intervallo.
Dopo l'applicazione della funzione non lineare avviene una normale quantizzazione uniforme.

4.3. Metodo di Lloyd-Max

Il metodo di Lloyd-Max (o metodo dei k-means) è un metodo iterativo di clustering molto utilizzato nel Machine Learning.
Tale metodo, dato un dataset, ha l'obbiettivo di trovare k centroidi (k è un iperparametro) che rappresentino al meglio il nostro dataset.
Nel caso della quantizzazione questi k centroidi sono i valori di ricostruzione che minimizzano l'errore di ricostruzione del dato.
Il numero di valori di ricostruzione scelto dipende dal numero di bits (in particolare avremo $2^{\text{BITS}} - 1$ valori di ricostruzione), pertanto risulta evidente come applicare un tale metodo computazionalmente molto pesante con $2^{16} - 1$ valori di ricostruzione possa non essere la migliore scelta.
Per maggiori dettagli del metodo di Lloyd-Max e alla sua implementazione nel nostro codice far riferimento ai commenti nelle funzioni presenti in *lloyd_max_quantizer.c*

4.4. Omomorfica

Come spiegato in precedenza, una quantizzazione omomorfica permette di maneggiare dati quantizzati senza bisogno di dequantizzarli.
Tale proprietà nel nostro codice è data da due fattori: un intervallo di quantizzazione unico e condiviso dagli array di ogni nodo e uno spazio di quantizzazione ridotto che permette l'addizione degli elementi senza incorrere in overflow.
Per quanto riguarda l'intervallo di quantizzazione unico, minimi e

massimi trovati da ogni processo vengono confrontati in una standard MPI_Allreduce con operatore MPI_Max (facciamo tutto in una sola chiamata su un array contenente -min, max, evitando quindi di dover fare due chiamate di riduzione con operatori diversi).
La riduzione dello spazio di quantizzazione viene fatta a seconda delle dimensioni di *comm_sz*, facendo quindi attenzione al poter accomodare nel worst-case-scendario una addizione fra *comm_sz* elementi di valore max.
Quella che avviene dopo è una normale quantizzazione uniforme.

4.5. Range Noto

Questo algoritmo non è altro che una versione dell'algoritmo di quantizzazione omomorfica nel quale però il range di rappresentazione dei valori è conosciuto a priori e uguale per tutti i processi (nel nostro caso utilizziamo semplicemente i valori passati al generatore di array).
Ovviamente tale algoritmo non è applicabile qualora questi range non si conoscano a priori, ma permette di risparmiare sia la ricerca di minimo e massimo, che la allreduce di tali valori.

5. RISULTATI

5.1. Speedup

I test sono stati effettuati facendo girare il programma variando le sue configurazioni su un cluster HPC dal professor Daniele De Sensi (che ringraziamo ancora per la disponibilità).
Di seguito il tempo impiegato al variare del numero dei nodi con *BITS=8*. *Tale scelta deriva dal fatto che è stato subito evidente dai nostri risultati come i vantaggi computazionali, già marginali nella quantizzazione a 8 bits, diventano nulli nel passaggio ai 16 bits*

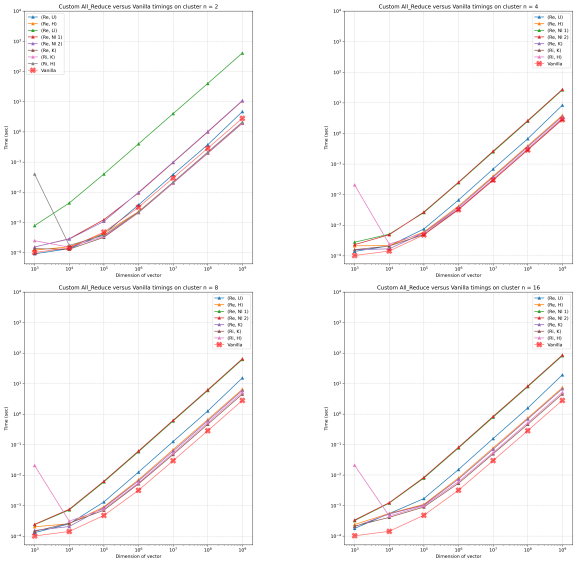

Figure 2: timings

La prima cosa che notiamo è che, a prescindere dal numero di nodi e dal tipo di struttura di riduzione, metodi computazionalmente intensivi come il Lloyd-Max (in verde) e le quantizzazioni non lineari (rosso e viola), risultino troppo lenti per essere utilizzati in questo ambito.
Al contrario vogliamo sottolineare come nella quantizzazione uniforme, e soprattutto nelle due quantizzazioni omomorfiche (ricordiamo che known_range è a tutti gli effetti una quantizzazione omomorfica), si riscontrino tempi simili, e in alcuni casi migliori, alla baseline.
Come facilmente pronosticabile, in quasi tutti i test effettuati, le quantizzazioni omomorfiche, soprattutto la known_range, sono le più veloci con entrambe le strutture di riduzione.
In particolare, soprattutto con due nodi, al crescere delle dimensioni del vettore notiamo un miglioramento prestazionale (a prescindere dal tipo di struttura di invio utilizzata).
Contemporaneamente notiamo come all'aumentare del numero di nodi questi vantaggi tendano a decrescere.
Tale miglioramento prestazionale all'aumentare della quantità di dati è presente anche a numeri di nodi maggiori, ma si nota come, anche con dimensioni molto grandi (>40MB) la baseline data da MPI_Allreduce non venga raggiunta.

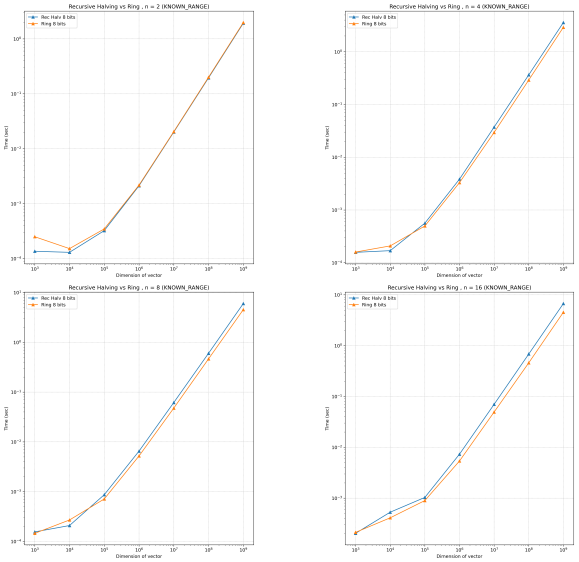

Figure 3: Ring vs RecHalving

Per quanto riguarda il tipo di struttura di quantizzazione, abbiamo trovato che la riduzione ad anello abbia dei vantaggi computazionali crescenti con l'aumentare dei nodi rispetto alla riduzione ad albero.

5.2. Errore di ricostruzione

Tra i dati ricostruiti in seguito alla quantizzazione incorre un naturale errore dovuto all'approssimazione dell'algoritmo scelto. Pertanto abbiamo misurato il *normalized mean square error*

$$NMSE = \frac{1}{N}\sum_{i=0}^{N}\frac{(x_i - y_i)^2}{x_i y_i}$$

Di seguito risultati ottenuti con la quantizzazione uniforme (con

struttura recursive_halving), generalizzabili alle differenze di errore riscontrate con altri algoritmi nel passare da 8 a 16 bits.

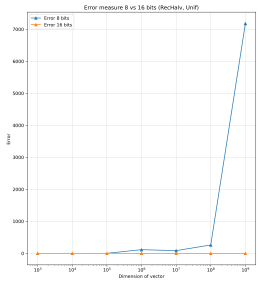

Figure 4: Reconstruction NMSE

Risulta chiaro come nel passare da un intervallo di quantizzazione di $2^8 - 1$ a uno di $2^{16} - 1$ ci sia una differenza abbissale in termini di precisione dei valori di ricostruzione, ma il costo computazionale e di invio pagato risulta eccessivo a tal punto da risultare uno svantaggio rispetto all'utilizzo della normale collective su dati non quantizzati.

6. CONCLUSIONI

Ci teniamo a sottolineare come i risultati ottenuti siano sicuramente dipendenti da implementazioni del codice non state-of-the-art, al contrario del codice della normale MPI_Allreduce.
Proprio alla luce di questo fatto, risulta quindi chiaro che gli effettivi vantaggi computazionali di questi metodi di quantizzazione siano probabilmente molto maggiori di quelli da noi evidenziati.
I grafici dimostrano risultati verificati anche in studi molto più sofisticati ed avanzati come [Communication Quantization for Data-Parallel Training of Deep Neural Networks] ovvero che l'overhead introdotto dagli algoritmi di quantizzazione nella maggior parte dei casi rende inutile il solo utilizzo di queste tecniche se non in combinazione con tecniche di sparsificazione.
Il paper appena citato analizza anche metodi di quantizzazione "estremi" fino a 1 o 2 bits, completamente estranei alla nostra ricerca e trova questo tipo di quantizzazioni le più vantaggiose nel training di network neurali.
Lavori di ricerca futuri potrebbero andare sia nella direzione di aggiungere questi metodi di quantizzazione, sia nell'utilizzo della sparisficazione.

7. REFERENCES

[1] https://github.com/lykelynothing/MultiCoreProject

[2] https://it.wikipedia.org/wiki/Omomorfismo

[3] https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=7835789.

[4] https://dlp-kdd.github.io/assets/pdf/a11-yang.pdf.