Calcolo parallelo e distribuito (CPD22)

Progetto 7a - LAR Congruence

Gabriele Romualdi (Matricola: 521111) Simone Chilosi (Matricola: 522155)

1 Concetti chiave

Link Repository Github: https://github.com/Panemiele/LarCongruence.jl

1.1 Matrici Sparse

In analisi numerica, una matrice sparsa è una matrice i cui valori sono quasi tutti uguali a zero. I pochi valori presenti sono distribuiti in modo casuale, cioè non si concentrano in determinate aree specifiche (Cluster).

1.2 Semiring

Un Semiring è una struttura algebrica che generalizza l'aritmetica reale rimpiazzando $(+,\cdot)$ con l'operazione binaria (Op1, Op2).
Un Semiring, in GraphBLAS, viene definito come l'unione di un **monoide M** e un **operatore binario moltiplicativo F**;

- Il monoide è una struttura algebrica formata da un operatore binario **associativo e commutativo di tipo additivo** e da **un dominio D** che deve contenere anche un elemento vuoto (simbolo dell'operatore: $\oplus$);
- L'operatore binario F è formato invece da **due domini di input e un dominio di output** (simbolo dell'operatore: $\otimes$).

1.3 SIMD

Single Instruction, Multiple Data (SIMD) è un metodo per parallelizzare i calcoli all'interno della CPU, per cui una singola operazione viene eseguita su più elementi di dati contemporaneamente. Le moderne architetture della CPU contengono set di istruzioni che possono farlo, operando su molti variabili contemporaneamente.
Questo non rende ogni ciclo più veloce. In particolare, si noti che l'utilizzo di SIMD implica che l'ordine delle operazioni all'interno e attraverso il ciclo potrebbe cambiare. Il compilatore deve essere certo che il riordino sia sicuro prima che tenti di parallelizzare un ciclo

1.4 Tasks

Un **Task** è semplicemente un insieme di istruzioni che possono essere sospese e riprese in qualsiasi momento all'interno di quell'insieme. Una funzione può anche essere pensata come un insieme di istruzioni, e quindi di attività può essere visto come qualcosa di simile. Ma ci sono due differenze fondamentali:

- Non c'è overhead per passare da un Task all'altro, il che significa che non viene riservato spazio nello stack per un cambio;
- a differenza di una funzione che deve terminare prima che il controllo torni al chiamante, un Task può essere interrotto e il controllo può essere passato a un altro Task in molti momenti diversi durante la sua esecuzione. In altre parole, nelle attività non esiste una relazione gerarchica chiamante-chiamato. Questo da l'impressione di lavorare in parallelo.

1.5 Threads

I **Thread** sono sequenze di calcolo che possono essere eseguite indipendentemente su un core della CPU, contemporaneamente ad altre sequenze simili.
A differenza dei task, che sono leggeri, i thread devono memorizzare uno stato quando vengono scambiati. Così, mentre si possono avere centinaia o migliaia di task in esecuzione, è opportuno avere solamente un numero limitato di Thread, tipicamente pari al numero di core della macchina in uso.

2 GraphBLAS

GraphBLAS è la libreria che offre funzioni per matrici sparse, la più comune quando si parla di calcolo parallelo e distribuito. Questa libreria offre metodi smart ed efficaci per memorizzare valori ed effettuare operazioni su di essi all'interno di matrici sparse.
Si può scegliere di implementare GraphBLAS sia sulla CPU che su GPU (la cosa interessante delle GPU è che lo si può fare in parallelo).
Si usano le matrici per rappresentare i grafi, in modo da poter utilizzare le operazioni dell'algebra lineare che sono molto veloci da eseguire: il prodotto fra matrici, per esempio, permette di ricavare informazioni sui percorsi possibili e nodi vicini.
GraphBLAS utilizza oggetti matematici chiamati "Semirings" che permettono di implementare qualsiasi operatore matematico e definire così un nuovo modello di prodotto matriciale. Due esempi:

1. Plus-times: tipico prodotto matriciale
2. Tropical Semiring: usa i seguenti operatori:
   - interno: la somma
   - esterno: il valore minimo

3 Local Congruence

E' una libreria che estende il funzionamento di GraphBLAS, oltre che ai grafi, anche ai complessi cellulari: l'obiettivo è verificare la congruenza di celle e complessi di celle.
Per calcolare la congruenza di complessi di catena locali si procede come segue:

- Per ogni faccia, si costruisce il suo complesso di catene locale, cioè la partizione del piano bidimensionale (identificato con z=0) indotta dal bordo di quella faccia e da tutte le altre che la dividono.
- Si rimette assieme per calcolare la congruenza: dall'insieme di complessi di catene locali ad ognuna delle facce dell'input, bisogna arrivare ad un unico complesso, "incollando" fra loro in modo coerente i vari complessi locali.

Si nota che l'operazione che può essere parallelizzata è la prima, questo perché viene eseguita in modo indipendente per ciascuna delle facce.

4 Studio Esecutivo - Ottimizzazioni

In questa sezione, verrà descritto il modo in cui sono stati modificate le funzioni dei progetti cea-SM.jl e cea-GB.jl, in modo tale da aumentare il grado di parallelismo e migliorare le performance.
Per consultare il codice, si rimanda al link del repository GitHub.

4.1 Ottimizzazioni per Sparse Matrix

Verranno mostrate le modifiche effettuate su cea-SM.jl, per produrre il nuovo file **cea-SM-optimized.jl**, e i risultati ottenuti, confrontando i tempi di esecuzioni delle due versioni.

4.1.1 Modifiche

Di seguito, verranno elencate le modifiche effettuate per migliorare l'esecuzione delle funzioni:

- la funzione **vertCongruence** (dal file verticesCongruence.jl) è stata modifica in **vertCongruenceOptimized** (nel file verticesCongruence-optimized.jl), aggiungendo l'utilizzo dei Tasks;
- la funzione **chainCongruenceSM_OPT(...)** è stata migliorata sfruttando i Tasks;
- la funzione **cellCongruenceSM_OPT(...)** è stata migliorata sfruttando i Tasks e SIMD.

4.1.2 Risultati

Per l'esecuzione dei test, per ottenere i vari risultati, sono stati utilizzati tre esempi:

- un primo esempio, con un input di dimensioni abbastanza contenute, con tempi di calcolo molto brevi;
- un secondo, che utilizza un input leggermente più grande del primo, ma con tempi comunque ridotti;
- un terzo esempio, che sfrutta un input molto grande e la cui esecuzione richiede un tempo nell'ordine dei secondi, considerevole rispetto ai primi due.

Per quanto riguarda il primo esempio, si è ottenuto un leggero miglioramento di circa $30\mu s$: questo però è dovuto al fatto che i tempi sono molto contenuti.
Nel secondo e terzo esempio, invece, è peggiorato di poco: anche quì, i tempi molto ristretti hanno fatto sì che aggiungere un grado di parallelismo più alto abbia aumentato l'overhead sulle funzioni, con un conseguente aumento del tempo di esecuzione (comunque un aumento molto basso).

4.2 Ottimizzazioni per GraphBLAS

Come per il caso di Sparse Matrix, si discuteranno le modifiche applicate a cea-GB.jl per produrre il nuovo file **cea-GB-optimized.jl**; inoltre, verranno discussi i risultati ottenuti.

4.2.1 Modifiche

Per quanto riguarda GraphBLAS, è stato scelto di implementare una sola modifica, per non penalizzare le performance: sono stati usati i Thread (in particolare 6) per eseguire la funzione **cellCongruence_OPT(...)**.

4.2.2 Risultati

Anche in questo caso, sono stati considerati tre esempi: il primo con un input di piccole dimensioni, il secondo con una dimensione dell'input medio-bassa e il terzo con un input molto grande.
Nonostante l'unica modifica, il risultato ottenuto è stato il migliore ottenuto, poiché aggiungere un'eccessiva parallelizzazione (con Tasks e SIMD) avrebbe comportato un overhead inutile; difatti, **nell'esempio 3**, aggiungendo i Tasks ad ogni ciclo "for" si è ottenuto un risultato pari a **96ms**, mentre, utilizzando solamente i Thread, sono stati raggiunti i **73ms**. In ogni caso, c'è stato una notevole diminuzione dei tempi di esecuzioni rispetto alla versione normale, che impiega circa **280ms**.
Per quanto riguarda i primi due esempi, i miglioramenti sono meno evidenti ma comunque presenti.

Riferimenti bibliografici

[1] *Hands-On Design Patterns and Best Practices with Julia Proven solutions to common problems in software design for Julia 1.x.* 2020.

[2] Avik Sengupta. *Julia High Performance : Optimizations, Distributed Computing, Multithreading, and GPU Programming with Julia 1. 0 and Beyond, 2nd Edition.* 2019.

[1] [2]