Progetto MIDTERM

Relazione sul progetto C++ di calcolo dell'istogramma dei bigrammi e dei trigrammi.

Introduzione:
Il progetto consiste nella creazione di un'applicazione C++ per calcolare l'istogramma dei bigrammi e dei trigrammi di un testo. L'istogramma rappresenta la frequenza di occorrenza dei bigrammi e dei trigrammi. L'applicazione supporta sia il calcolo in modo sequenziale che in modo parallelo utilizzando l'API OpenMP.

Implementazione:
Il codice è suddiviso in diverse funzioni: Le funzioni `computeCharBigramHistogram` e `computeCharTrigramHistogram` calcola l'istogramma dei bigrammi e dei trigrammi dei caratteri , mentre le funzioni `computeWordBigramHistogram` e 'computeWordTrigramHistogram' calcolano l'istogramma dei bigrammi e dei trigrammi delle parole. Nel codice le funzioni vengono ripetute aggiungendo 'Sequential' o 'Parallel' per distinguere le sezioni sequenziali da quelle parallele.
Le funzioni parallele implementano il calcolo parallelo degli istogrammi utilizzando OpenMP. Ogni thread esegue un'iterazione su porzioni distinte del testo, calcolando l'istogramma locale dei bigrammi/trigrammi di caratteri/parole. Successivamente, gli istogrammi locali vengono aggregati in un unico istogramma globale. L'uso di #pragma omp atomic è fatto per garantire la correttezza durante l'aggiornamento dell'istogramma globale.
Nel `main`, il testo viene letto da un file di testo nella directory corrente chiamato "generic_text.txt", contenete testo casuale per un ammontare di circa 100MB. Dopo che il testo viene processato dalle 4 funzioni sequenziali, i risultati vengono stampati a schermo insieme al tempo di esecuzione.
Successivamente viene eseguita una sezione parallela per calcolare gli istogrammi sia con 16, 8 che 4 thread. Il numero di thread viene impostato utilizzando la funzione omp_set_num_threads(). La direttiva '#pragma omp parallel sections' definisce una sezione parallela, all'interno della quale diverse operazioni possono essere suddivise tra i thread. Ogni sezione è marcata da #pragma omp section, e ogni sezione può essere eseguita in parallelo. Le diverse esecuzioni vengono misurate in termini di tempo e viene calcolato il relativo speedup rispetto all'esecuzione sequenziale. Utilizzando #pragma omp section, ogni sezione viene eseguita da un sottoinsieme dei thread disponibili nel team parallelo. Ciò permette di massimizzare il parallelismo, consentendo a diverse operazioni

indipendenti di procedere simultaneamente grazie all'assegnazione di sezioni a thread diversi.
Quando un team di thread accede ad una funzione, vengono inizializzate le seguenti variabili:

- histogram: Mappa che conterrà l'istogramma finale dei bigrammi delle parole.
- loc: Oggetto di tipo std::locale utilizzato per la manipolazione delle stringhe.
- numThreads: Numero di thread paralleli, ottenuto mediante la funzione omp_get_num_threads().
- textLength: Lunghezza totale del testo in input.
- localHistograms: Un vettore di mappe, ognuna delle quali rappresenta un istogramma locale gestito da un thread parallelo.

  #pragma omp parallel num_threads(numThreads): Inizia una sezione parallela con il numero specificato di thread.
- Calcolo della porzione di testo che ciascun thread deve processare in modo parallelo.

Alla fine della sezione parallela, tutti gli istogrammi locali vengono aggregati in modo sicuro in un'unica mappa globale utilizzando un'operazione atomica.

Utilizzo del framework Sanitizer tramite WSL (Windows Subsystem for Linux)
Nel mio progetto, ho avuto la necessità di utilizzare il framework Sanitizer per migliorare la sicurezza del mio codice C++. Ho quindi adottato un approccio alternativo per eseguirlo su un ambiente Windows. Per fare ciò, ho utilizzato WSL (Windows Subsystem for Linux). WSL è un ambiente di compatibilità fornito da Microsoft che mi ha permesso di eseguire un'istanza completa di un sistema operativo Linux sul mio sistema Windows. Dopo aver installato WSL e una distribuzione Linux (Kali) ho utilizzare il framework Sanitizer attivando l'opzione -fsanitize durante la compilazione del mio codice.

Risultati:
I risultati stampati a schermo includono gli istogrammi dei bigrammi e dei trigrammi ottenuti sia dall'esecuzione sequenziale che da quella parallela. Inoltre, vengono visualizzati i tempi di esecuzione per entrambe le modalità e il valore del speedup. I risultati vengono però mostrati per la lettura di un file più piccolo, chiamato "generic_books" dal peso complessivo di circa 14MB poiché i test sul file precedentemente richiedevano dei tempi di attesa significativi a causa delle scarse prestazioni della mia CPU.

Conclusioni:

Il progetto dimostra come l'utilizzo di OpenMP ha migliorato le prestazioni di calcolo degli istogrammi dei bigrammi e dei trigrammi. L'esecuzione parallela consente di sfruttare la potenza di calcolo multi-threaded della CPU, riducendo il tempo di esecuzione complessivo rispetto all'esecuzione sequenziale. Ho notato soprattutto che nonostante la mia CPU disponga di soli 4 core fisici, l'utilizzo di OpenMP e la creazione di 16 o 8 threads virtuali durante l'esecuzione parallela offrono un notevole beneficio in termini di prestazioni del programma. La parallelizzazione del calcolo degli istogrammi consente ai threads virtuali di coesistere efficacemente sulla stessa sezione hardware, sfruttando le risorse disponibili in modo più efficiente. Questo risultato è evidenziato dai tempi di esecuzione più brevi ottenuti con un numero maggiore di threads rispetto all'esecuzione con 4 thread. L'uso del parallelismo, anche su un numero di core fisici inferiore, dimostra la capacità di migliorare le prestazioni complessive del programma, fornendo un'implementazione efficiente e ottimizzata per sfruttare al meglio le risorse hardware disponibili. Ricavando alcune informazioni dalle slide e da forum online, si può trarre questa conclusione: il concetto di threading nei sistemi operativi offre la possibilità di eseguire istruzioni in modo parallelo, consentendo ai programmi di sfruttare il tempo di attesa per risorse esterne. I thread, che possono essere visti come contesti di esecuzione con un proprio storico e dati specifici, sono utilizzati per gestire attività simultanee in un programma. Quando si dispone di un processore con più core fisici, l'operating system può allocare i thread in modi diversi, cercando di ottimizzare l'esecuzione.
L'introduzione del concetto di "hyper-threading" cerca di ottimizzare ulteriormente l'utilizzo dei core fisici, consentendo loro di lavorare più velocemente riducendo la necessità di cambiare contesto frequentemente. Questo non implica la presenza di core fisici aggiuntivi, ma piuttosto sfrutta in modo più efficiente i core esistenti. Nonostante il fatto che un processore possa avere un numero limitato di core fisici, l'utilizzo di "hyper-threading" può ancora portare a miglioramenti delle prestazioni, consentendo una gestione più efficiente delle istruzioni e minimizzando il tempo dedicato allo switch di contesto. In ultima analisi, l'efficacia del threading e del "hyper-threading" dipende dall'implementazione specifica del processore e dal supporto fornito dal sistema operativo.
