Risoluzione di Sistemi Lineari di Grandi Dimensioni Utilizzando Calcolo Parallelo su Cluster HPC

Dimonte Giuseppe
Università degli Studi di Parma
giuseppe.dimonte@studenti.unipr.it
Matricola: 367431

Abstract

Questo progetto descrive l'implementazione e l'esecuzione di algoritmi paralleli per la risoluzione di sistemi lineari di grandi dimensioni utilizzando il metodo di Gauss-Seidel su un cluster di calcolo ad alte prestazioni (HPC) presso l'Università di Parma (UniPr). Vengono esplorati tre approcci differenti: MPI, OpenMP e CUDA, per sfruttare al meglio le risorse del cluster HPC.

Contents

1 Introduzione

La risoluzione di sistemi lineari di grandi dimensioni è una delle problematiche fondamentali nel campo del calcolo scientifico. Questo progetto mira a implementare il metodo di Gauss-Seidel in modo parallelo utilizzando tre differenti tecnologie: MPI per il calcolo distribuito, OpenMP per il calcolo parallelo su CPU e CUDA per il calcolo parallelo su GPU.

2 Generazione del Dataset

Per testare gli algoritmi, è stato generato un dataset sintetico utilizzando uno script Python. Il dataset consiste in una matrice A di grandi dimensioni e un vettore b. La matrice A è stata generata in modo tale da essere diagonale dominante per garantire la convergenza del metodo di Gauss-Seidel.

- **Generazione della matrice** A: La matrice è generata casualmente e resa diagonale dominante, dove ogni elemento diagonale è maggiore della somma degli elementi non diagonali nella stessa riga.
- **Generazione del vettore** b: Il vettore è generato casualmente.
- **Salvataggio dei dati**: La matrice A e il vettore b sono salvati in file di testo per essere letti dagli algoritmi implementati.

3 Tecnologie Utilizzate

3.1 MPI

Message Passing Interface (MPI) è un protocollo di comunicazione utilizzato per il calcolo parallelo su sistemi distribuiti. MPI permette ai processi di comunicare tra loro mediante l'invio e la ricezione di messaggi, ed è particolarmente efficace per eseguire calcoli su cluster di computer.

3.2 OpenMP

Open Multi-Processing (OpenMP) è un'API per la programmazione parallela su sistemi a memoria condivisa. OpenMP utilizza direttive di compilazione per suddividere i compiti tra più thread eseguiti su diverse CPU di un singolo nodo. È particolarmente utile per sfruttare i processori multi-core moderni, consentendo la parallelizzazione del codice in modo semplice e intuitivo.

3.3 CUDA

Compute Unified Device Architecture (CUDA) è una piattaforma di calcolo parallelo e un modello di programmazione sviluppato da NVIDIA per il calcolo su GPU. CUDA permette di eseguire calcoli intensivi in parallelo sfruttando migliaia di core presenti sulle GPU, rendendolo ideale per accelerare applicazioni scientifiche e ingegneristiche.

4 Descrizione dei Vantaggi

4.1 MPI

- **Scalabilità**: Ideale per sistemi con un gran numero di nodi, come supercomputer e cluster di grandi dimensioni.
- **Flessibilità**: Funziona su una varietà di architetture, incluso cluster eterogenei.
- **Efficienza**: Comunicazione diretta tra processi, riducendo la latenza.
- **Portabilità**: Standard ampiamente adottato disponibile su molte piattaforme.

4.2 OpenMP

- **Facilità d'uso**: Parallelizzazione semplice tramite direttive di compilazione.
- **Efficienza in memoria condivisa**: Ideale per sistemi con memoria condivisa come workstation multi-core.
- **Flessibilità**: Controllo granulare sulla parallelizzazione e ottimizzazione delle prestazioni.
- **Compatibilità**: Funziona con codice C, C++ e Fortran esistente.

4.3 CUDA

- **Potenza di calcolo**: Sfrutta migliaia di core GPU per un'enorme accelerazione del calcolo.
- **Efficienza energetica**: GPU ottimizzate per calcoli intensivi con consumo energetico ridotto.

- **Librerie ottimizzate**: Supporto per librerie matematiche e scientifiche altamente ottimizzate.
- **Parallelismo fine**: Capacità di gestire migliaia di thread simultaneamente per calcoli complessi.

5 Implementazione

5.1 Implementazione MPI

L'implementazione MPI del metodo di Gauss-Seidel prevede l'utilizzo della libreria MPI per la comunicazione tra processi. Di seguito vengono descritti i passaggi principali dell'algoritmo.

- **Inizializzazione MPI**: Viene inizializzato l'ambiente MPI con `MPI_Init`, ottenendo il rank del processo corrente e il numero totale di processi.
- **Divisione del lavoro**: La matrice A e il vettore b vengono suddivisi in parti uguali tra i processi utilizzando `MPI_Scatter`. Ogni processo riceve un sottoinsieme delle righe della matrice e del vettore.
- **Iterazioni di Gauss-Seidel**: Per ogni iterazione, ogni processo calcola i nuovi valori del proprio sottoinsieme del vettore soluzione x. La somma dei prodotti esclusi dalla diagonale viene calcolata e sottratta dal termine noto, poi divisa per l'elemento diagonale.
- **Comunicazione tra processi**: I nuovi valori calcolati dai processi vengono comunicati a tutti gli altri processi utilizzando `MPI_Allgather`, in modo che ogni processo abbia una copia aggiornata del vettore soluzione x.
- **Controllo della convergenza**: Viene calcolata la norma del residuo locale per ogni processo e i risultati vengono ridotti a una somma globale utilizzando `MPI_Allreduce`. Se la norma globale è al di sotto di una certa soglia di tolleranza, l'iterazione si interrompe.
- **Pulizia e finalizzazione**: Alla fine dell'algoritmo, la memoria allocata viene liberata e l'ambiente MPI viene chiuso con `MPI_Finalize`.

5.2 Implementazione OpenMP

L'implementazione OpenMP sfrutta la parallelizzazione su CPU tramite direttive di compilazione. I passaggi principali sono i seguenti:

- **Inizializzazione dei dati**: La matrice A e i vettori b e x vengono inizializzati.
- **Iterazioni di Gauss-Seidel**: Utilizzando la direttiva `#pragma omp parallel for`, il ciclo principale dell'algoritmo viene parallelizzato. Ogni thread calcola i nuovi valori del vettore x per un sottoinsieme delle righe della matrice.
- **Controllo della convergenza**: La convergenza viene verificata calcolando la norma del residuo e interrompendo l'iterazione se la norma è inferiore a una soglia di tolleranza.
- **Pulizia**: Alla fine dell'algoritmo, la memoria allocata viene liberata.

5.3 Implementazione CUDA

L'implementazione CUDA sfrutta la parallelizzazione su GPU per accelerare il calcolo. I passaggi principali sono i seguenti:

- **Allocazione e copia dei dati**: La matrice A e i vettori b e x vengono copiati dalla memoria dell'host alla memoria del dispositivo (GPU) utilizzando `cudaMalloc` e `cudaMemcpy`.
- **Kernel di Gauss-Seidel**: Un kernel CUDA viene lanciato per calcolare i nuovi valori del vettore x in parallelo su più thread. Ogni thread calcola il valore per un singolo elemento del vettore.
- **Iterazioni**: Il kernel viene lanciato ripetutamente per il numero massimo di iterazioni o fino al raggiungimento della convergenza.
- **Copia dei risultati**: I risultati vengono copiati dalla memoria del dispositivo alla memoria dell'host utilizzando `cudaMemcpy`.
- **Pulizia**: La memoria allocata sulla GPU viene liberata con `cudaFree`.

6 Conclusione

Questo progetto ha dimostrato come l'algoritmo di Gauss-Seidel possa essere parallelizzato utilizzando tre diverse tecnologie: MPI, OpenMP e CUDA. Ogni tecnologia offre vantaggi specifici a seconda dell'architettura hardware disponibile e delle caratteristiche del problema da risolvere. Utilizzando il cluster HPC dell'Università di Parma, è stato possibile sfruttare al massimo le capacità di calcolo parallelo per risolvere sistemi lineari di grandi dimensioni in modo efficiente.

7 Conclusione

Questo progetto ha dimostrato come l'algoritmo di Gauss-Seidel possa essere parallelizzato utilizzando tre diverse tecnologie: MPI, OpenMP e CUDA. Ogni tecnologia offre vantaggi specifici a seconda dell'architettura hardware disponibile e delle caratteristiche del problema da risolvere. Utilizzando il cluster HPC dell'Università di Parma, è stato possibile sfruttare al massimo le capacità di calcolo parallelo per risolvere sistemi lineari di grandi dimensioni in modo efficiente.

8 Bibliografia

References

[1] P. S. Pacheco, *An Introduction to Parallel Programming with MPI*, Morgan Kaufmann, 2011.

[2] B. Chapman, G. Jost, and R. van der Pas, *Using OpenMP: Portable Shared Memory Parallel Programming*, MIT Press, 2008.

[3] J. Sanders and E. Kandrot, *CUDA by Example: An Introduction to General-Purpose GPU Programming*, Addison-Wesley Professional, 2010.

[4] Centro di Calcolo dell'Università di Parma, *HPC Cluster User Guide*, 2024. [Online]. Available: https://www.hpc.unipr.it/dokuwiki/doku.php?id=calcoloscientifico:userguide

[5] Il progetto in tutte le sue parti è stato caricato al seguente link : https://github.com/sh4nk7/Risoluzione-di-Sistemi-Lineari-di-Grandi-Dimensioni-