Progetto di High Performance Computing 2019/2020

Ivan Perazzini, matr 0000825047

2 maggio 2020

Introduzione

Il lavoro di ottimizzazione è stato svolto con l'obiettivo di apportare modifiche meno invasive possibili al programma di base.
La strategia principale adottata è stata quella di suddividere il carico di lavoro del ciclo a righe *252-256* del file fornito dal Professore fra le unità di lavoro di modo che ognuna trovasse il suo ottimo locale per poi controllare quale fra gli ottimi locali fosse quello globale, quindi assegnare next al punto ottimo globale.
Per poter parallelizzare l'applicazione, ho individuato e successivamente gestito le seguenti dipendenze nel ciclo a righe *244-259*:

- L'inserimento di un punto nell'array hull dipende dalla variabile cur **(RAW)**
- L'aggiornamento di next prima del ciclo for dipende dall variabile cur **(RAW)**
- L'aggiornamento di next nel ciclo for dipende da cur e da next stessa, che potrebbe essere modificata concorrentemente **(RAW e WAW)**
- L'aggiornamento di cur dipende dal valore finale di next al termine del ciclo for **(RAW)**

Esecuzione dei test sul programma

Per calcolare tempi di esecuzione e misure di speedup ed efficienza i programmi sono stati eseguiti sul server isi-raptor ciascuno cinque volte per numero di unità di lavoro (da 1 a 12) sul file di input "circ100k.in", essendo questo

l'insieme di dati che:
1) richiedeva tempi di elaborazione più lunghi e stabili.
2) dati i tempi di esecuzione lunghi della versione seriale, evidenziava meglio i vantaggi di una soluzione parallela.
Test meno approfonditi sugli altri casi hanno portato comunque a tendenze simili in termini di efficienza e speedup; nei casi con input che beneficiano meno dalla parallelizzazione del programma ovviamente i due valori risultano molto meno ottimali.

Versione MPI

Nell'implementazione MPI il vettore dei punti, dopo essere stato distribuito fra tutti i processi tramite broadcast iniziato dal processo 0, viene virtualmente suddiviso fra i processi.
Per semplificare il codice, infatti, ogni processo avrà una copia di tutti i punti, ma ne considererà solo una parte per la ricerca del prossimo punto dell'hull. Il punto iniziale dell'hull (quello più a sinistra fra tutti i punti) viene ricercato localmente da ogni processo nel proprio intervallo di punti da considerare; successivamente, tramite MPI_Allgather, tutti i processi ricevono tutti i punti locali trovati, e ne identificano quello globalmente più a sinistra.
In maniera simile, ogni processo troverà il suo punto ottimo locale e successivamente confronterà tutti i candidati ad essere il punto "next".
Essendo queste le uniche operazioni di comunicazione tra processi presenti nel programma ed avvenendo fuori dal ciclo for (zona dell'applicazione dove normalmente si concentra la maggior parte del lavoro), il tempo passato in fase di sincronizzazione fra i processi risulta essere basso, come dimostrato dai discreti valori di efficienza riportati in seguito.

Per ottimizzare ulteriormente l'applicazione ho anche aggiunto una struttura per memorizzare quali punti appartenessero all'hull, così da poterli saltare durante la ricerca del punto "next".
Questa ottimizzazione è stata tentata anche per la versione OpenMP ma ha portato a risultati che verranno discussi in seguito.

Figura 1: Speedup versione MPI

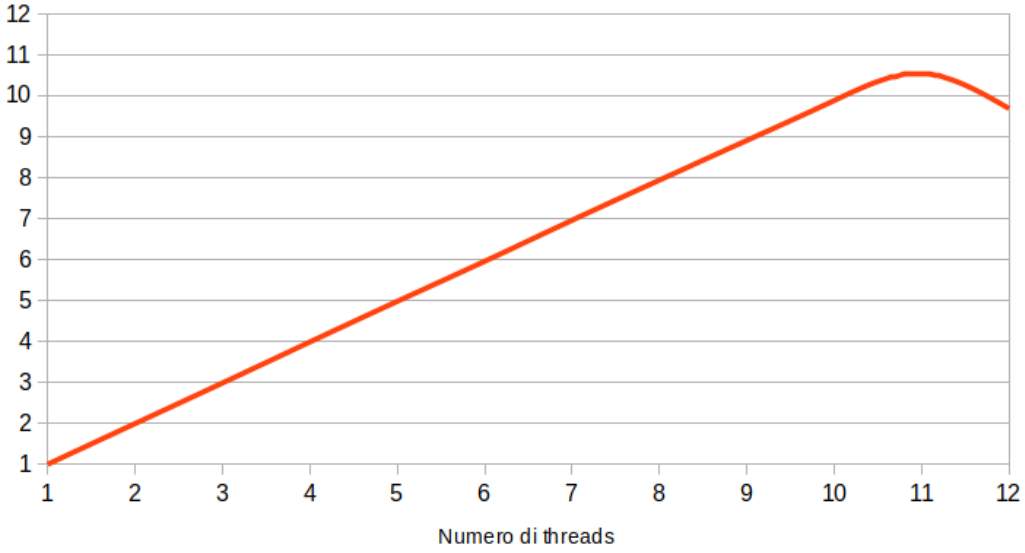


Figura 2: Efficienza versione MPI

Versione OpenMP

I due cicli presenti nella funzione da ottimizzare sono stati racchiusi dentro una sezione parallela, aperta solo una volta per non dover creare e distruggere continuamente il pool di thread ad ogni esecuzione del ciclo for.

Per contenere e aggiornare gli ottimi locali, ho creato un array di tanti elementi quanti sono i thread chiamato local_next e l'ho reso condiviso.
Ogni thread quindi, durante la ricerca del prossimo punto effettuata nel ciclo parallelo for, salva il punto trovato nella cella di local_next relativa al suo thread id.
Terminata la fase della ricerca degli ottimi locali, un solo thread fra tutti effettua un controllo seriale per trovare quali fra i punti in local_next sia quello da aggiungere all'hull.
Tutti gli aggiornamenti e letture delle variabili hull e cur sono stati effettuati dentro sezioni single: le due variabili, infatti, sono state rese condivise fra i thread per mantenerli sincronizzati.

Stimare le prestazioni della versione OpenMP è risultato più impegnativo di quella MPI, in quanto l'ottimizzazione effettuata dal compilatore gcc risulta estremamente influente sui parametri di efficienza e speedup.
Ho pertanto testato quattro versioni e mostrato i risultati nei grafici a seguire:

- no tag: versione senza ottimizzazione del compilatore e senza l'accorgimento di evitare di considerare punti già presenti nell'hull.
- no tag (O3): come sopra, ma con flag di ottimizzazione -O3 inserito nelle opzioni di compilazione.
- tag: versione senza ottimizzazione del compilatore ma con accorgimento sui punti già inseriti nell'hull.
- tag (O3): versione con accorgimento e ottimizzazione -O3

Figura 3: Speedup versioni OpenMP

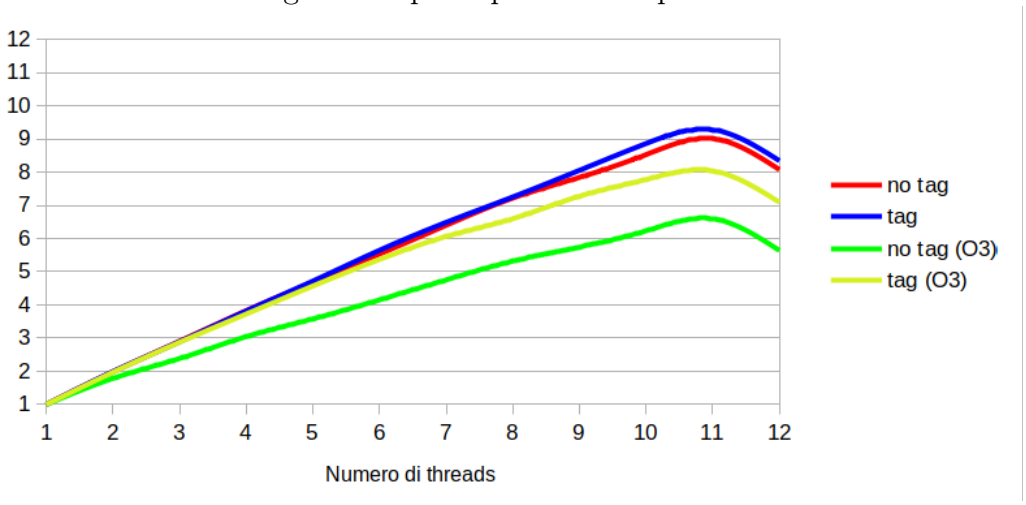


Figura 4: Efficienza versioni OpenMP

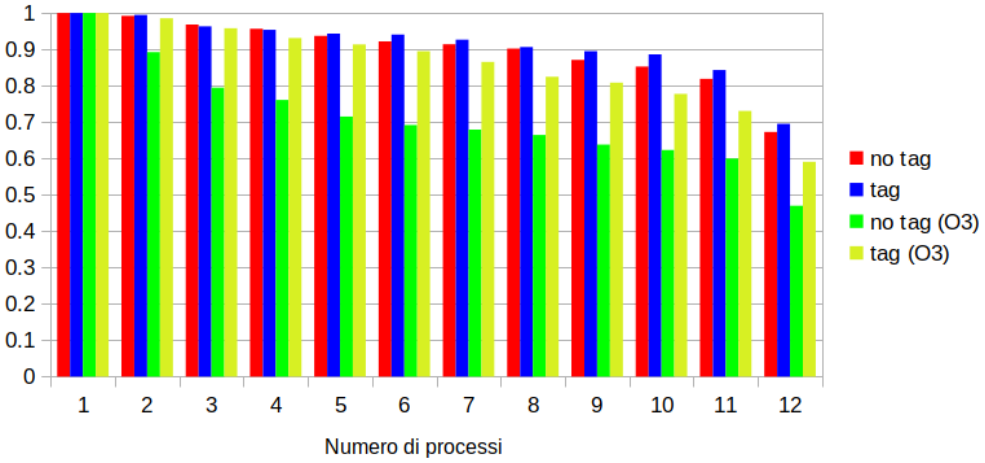
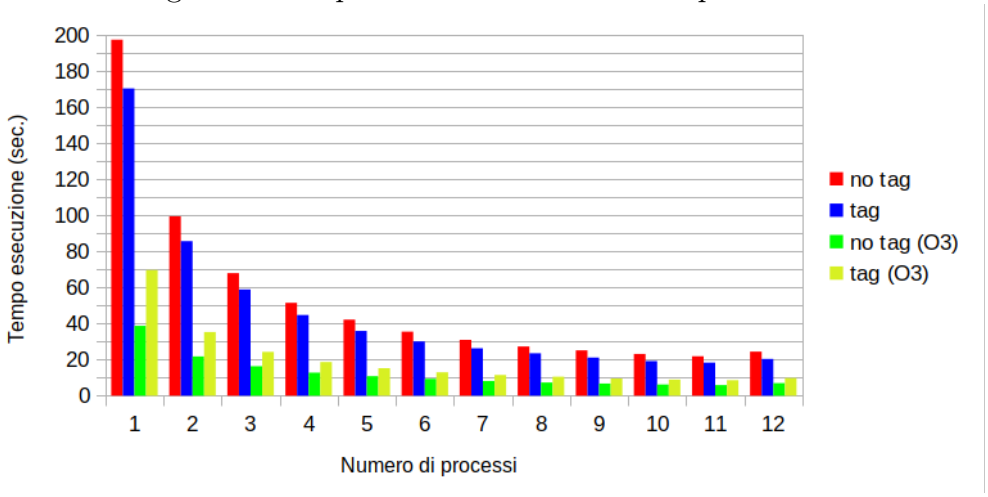


Figura 5: Tempi di esecuzione versioni OpenMP



(Nota: i discorsi seguenti sulle tempistiche si riferiscono solamente all'esecuzione del programma sulla macchina server isi-raptor)
Come si può vedere, senza l'aggiunta di parametri di ottimizzazione, la soluzione scala molto bene e, come atteso, l'aggiunta del controllo sui punti già inseriti nell'hull migliora generalmente le prestazioni.
Tuttavia, come evidenziato dal grafico dei tempi di esecuzione, l'aggiunta di ottimizzazione risulta estremamente conveniente dal punto di vista della velocità del programma.
Aggiungendo l'ottimizzazione però si verifica una situazione inaspettata: la

soluzione senza accorgimento risulta costantemente più veloce di quella con controlli aggiuntivi. A questa velocità però va contro una grande perdita di scalabilità, visto che la versione con controlli si mantiene costantemente almeno il 10% più efficiente. Nonostante la minore velocità di esecuzione (es: con un file di input "circ200k" la differenza minore fra i tempi arriva anche a 10 secondi), per quanto riguarda la consegna dell'elaborato ho deciso di dare priorità a efficienza e scalabilità e quindi consegnare la versione "tag (O3)", considerando che i suoi tempi di esecuzione si avvicinano comunque a quelli della versione MPI.

Conclusioni

Per quanto riguarda le prestazioni, la versione OpenMP proposta risulta comunque meno scalabile di quella MPI, indipendentemente dall'intervento del compilatore.
Ipotizzo che questo sia dovuto al fatto di non essere stato in grado di eliminare la dipendenza da "next" fra un iterazione e l'altra del ciclo esterno: questo porta la maggior parte dei thread a dover attendere che un'iterazione sia terminata prima di mettersi al lavoro sull'altra, non facendo nulla nel frattempo. Oltre a questo, sono anche presenti sezioni nelle quali opera un solo thread, diminuendo ulteriormente il lavoro realmente parallelizzato.
Un altro dei fattori svantaggiosi per la versione OpenMP è quello dell'accesso a memoria: il vettore di punti della figura, condiviso fra tutti i thread, è infatti acceduto concorrentemente in modo potenzialmente molto sparso.
L'implementazione con MPI risulta invece più performante, forse perché le operazioni di comunicazione necessarie per la sincronizzazione fra i processi e la preparazione di un nuova nuova iterazione (MPI_Gather ed MPI_Broadcast, nello specifico), essendo "collettive", tengono impegnati tutti i processi. A questo si aggiunge chiaramente, come sottolineato nella sezione precedente, il fatto che i processi non si trovano in condizione di dover leggere e scrivere su aree di memoria condivise e accedute in modo irregolare.
Ciò nonostante, è comunque ben visibile il limite utile di unità di calcolo per entrambe le implementazioni una volta raggiunti i dodici thread/processi.