Progetto di High Performance Computing 2023/2024

Alessandro Monticelli, matr. 0001028456
11/07/2024

Introduzione

L'obiettivo del progetto è quello di parallelizzare un programma seriale (già fornito) allo scopo di valutarne l'incremento prestazionale.

Al fine di eseguire il codice in parallelo sono state impiegate due differenti tecnologie di parallelizzazione: OpenMP ed MPI; permettendo di effettuare un confronto dell'efficacia di entrambe nella risoluzione del problema.

OpenMP permette di parallelizzare porzioni di codice attraverso direttive `#pragma` e alcune funzioni di libreria. Con una sintassi abbastanza semplice, è quindi possibile dividere il programma in regioni seriali e parallele.

MPI è una libreria per il calcolo parallelo distribuito, che si basa sulla comunicazione e la sincronizzazione tra processi indipendenti, che può eseguire su nodi separati o su *core* differenti in un sistema distribuito.

Versione OpenMP

Problemi riscontrati

1. **Eliminazione delle *loop-carried dependencies* all'interno dei cicli for.**

   Nella versione seriale del codice, l'inizializzazione dell'indice del ciclo interno dipendeva dal valore dell'indice del ciclo esterno.

   ```
   for i := 0 to N-1 do:
         for j := i+1 to N-1 do:
            // ...
         endfor
   endfor
   ```

   Per poter eliminare tale dipendenza e permettere la parallelizzazione del ciclo annidato, la struttura del ciclo interno è stata modificata come segue:

   ```
   for i := 0 to N-1 do:
         for j := 0 to N-1 do:
            if j > i do:
               // ...
            endif
         endfor
   endfor
   ```

2. **Parallelizzazione dei cicli for annidati.**

   OpenMP permette di parallelizzare due cicli for annidati attraverso la clausola `omp for collapse(2)`. Utilizzando questo approccio i due cicli for vengono accorpati in un unico spazio di iterazione, che viene poi suddiviso tra i thread del pool creato per l'esecuzione della regione parallela.

3. **Eliminazione delle *race conditions*.**

   Nonostante la rimozione della *loop-carried dependency* (1.), all'interno del codice sono ancora presenti operazioni che potrebbero causare corse critiche. In particolare due *thread* potrebbero tentare di modificare la posizione di una stessa coppia di cerchi contemporaneamente. Si elimina questo pericolo usando delle direttive `#pragma atomic` necessarie a garantire che le operazioni di modifica della posizione siano effettuate atomicamente.

4. **Politica di scheduling da utilizzare.**

   Per decidere quale politica di *scheduling* sia la più adatta è necessario determinare se il carico di lavoro resti costante o diminuisca durante l'esecuzione di iterazioni successive dei cicli for parallelizzati (2.).
   Si vede che la posizione dei cerchi viene modificata solo se la distanza tra i centri dei due cerchi è minore di un *EPSILON*. Possiamo assumere che con l'avanzare delle iterazioni il numero di sovrapposizioni di cerchi diminuisca, e quindi il carico di lavoro per ogni iterazione è decrescente. In questo caso si può preferire una politica di *scheduling* dinamica. La scelta risulta ottimale anche in base ai benchmark svolti usando le diverse politiche di *scheduling* messe a disposizione da OpenMP (*static, dynamic, guided*).

Prestazioni

L'incremento di prestazioni è stato misurato utilizzando 3 metriche: *Speedup*, *Strong Scaling Efficiency* e *Weak Scaling Efficiency*.

Speedup e Strong Scaling Efficiency

Per il calcolo dello *Speedup* e della *Strong Scaling Efficiency* sono state effettuate 5 prove per ogni dimensione del *pool* di *thread* (da 1 a 12), con un input di 5000 cerchi e 100 iterazioni. Per ogni prova si sono raccolti i tempi di esecuzione del programma (*t1,t2,t3,t4,t5*) con *p* processi, ottenendo il valore medio del *Wall Clock Time* (WCT) per ogni dimensione del *pool*. Indicando il WCT per n *thread* con **T(n)** e considerando **T(1) ≈ Tserial** si ottengono le seguenti formule per calcolare lo *Speedup* **S(p)** e la *Strong Scaling Efficiency* **E(p)**:

$$S(p)=\frac{T(1)}{T(p)}$$

Formula 1: Speedup

$$E(p)=\frac{S(p)}{p}$$

Formula 2: Strong Scaling Efficiency

p	t1	t2	t3	t4	t5	Average WCT	Speedup	Strong Scaling Efficiency
1	36.21	36.21	36.22	36.22	36.21	36.21	1.00	1.00
2	18.58	18.61	18.60	18.63	18.63	18.61	1.95	0.97
3	12.67	12.67	12.59	12.64	12.59	12.63	2.87	0.96
4	9.59	9.60	9.60	9.62	9.54	9.59	3.78	0.94
5	7.80	7.81	7.86	7.81	7.85	7.82	4.63	0.93
6	6.66	6.70	6.61	6.68	6.66	6.66	5.44	0.91
7	5.85	5.86	5.85	5.85	5.85	5.85	6.19	0.88
8	5.17	5.17	5.17	5.16	5.17	5.17	7.01	0.88
9	4.74	4.74	4.75	4.74	4.74	4.74	7.64	0.85
10	4.30	4.31	4.29	4.30	4.30	4.30	8.42	0.84
11	4.05	4.02	4.05	4.05	4.04	4.04	8.96	0.81
12	4.04	3.92	4.04	3.95	3.92	3.98	9.11	0.76

Tabella 1: Calcolo Speedup e Strong Scaling Efficiency

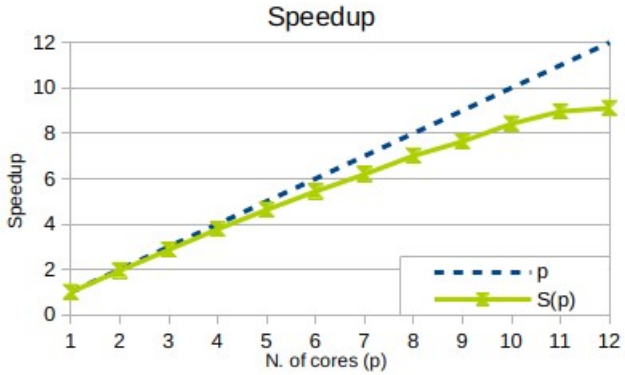


Grafico 1: Grafico Speedup OpenMP

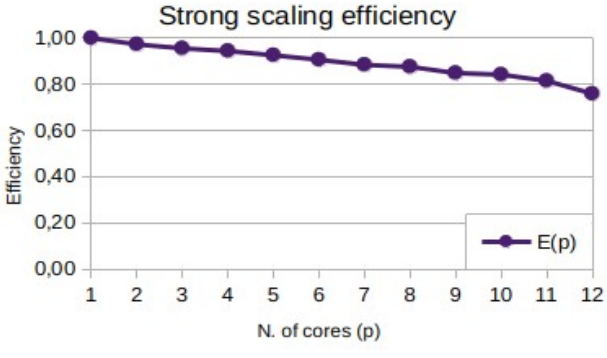


Grafico 2: Grafico SSE OpenMP

Weak Scaling Efficiency

L'obiettivo è misurare l'efficienza di esecuzioni che mantengano un carico costante per ogni *core*. Per una grandezza del problema **np** si vuole mantenere il lavoro eseguito da un singolo *core* **f(np,p)** costante. L'algoritmo implementato ha un costo computazionale di $\mathbf{O(n^2)}$, quindi:

$$n_p^2 / p = k \Rightarrow n_p = \sqrt{p}\,k$$

Formula 4: Carico di lavoro per thread

$$f(n_p^2, p) = k$$

Formula 3: Funzione del carico di lavoro per thread

$$n_p = \sqrt{p}\,k'$$

Formula 5: Carico di lavoro per thread

Sono state effettuate 5 prove per ogni dimensione del *pool* di processi (da 1 a 12), con un input (**n**) di base di 1000 cerchi e 100 iterazioni.

Per ogni prova si sono raccolti i tempi di esecuzione (*t1,t2,t3,t4,t5*), ottenendo il valore medio del *Wall Clock Time* (WCT) per ogni dimensione del *pool* e dell'input. La dimensione dell'input cresce in funzione alla dimensione del *pool* di *thread*, secondo la legge descritta in precedenza (Formula 5).

n	p	t1	t2	t3	t4	t5	Average WCT	Weak Scaling Efficiency
1,000	1	1.47	1.47	1.47	1.47	1.46	1.47	1.00
1,414	2	1.56	1.52	1.55	1.55	1.55	1.55	0.95
1,732	3	1.57	1.59	1.59	1.59	1.59	1.59	0.92
2,000	4	1.56	1.61	1.58	1.60	1.62	1.59	0.92
2,236	5	1.66	1.65	1.67	1.67	1.63	1.65	0.89
2,449	6	1.69	1.70	1.61	1.68	1.65	1.67	0.88
2,645	7	1.74	1.74	1.72	1.74	1.74	1.74	0.85
2,828	8	1.77	1.77	1.77	1.78	1.77	1.77	0.83
3,000	9	1.81	1.81	1.81	1.81	1.81	1.81	0.81
3,162	10	1.85	1.85	1.85	1.85	1.85	1.85	0.79
3,316	11	1.93	1.92	1.93	1.92	1.91	1.92	0.76
3,464	12	2.10	2.14	2.04	1.99	2.13	2.08	0.71

Tabella 2: Calcolo Weak Scaling Efficiency

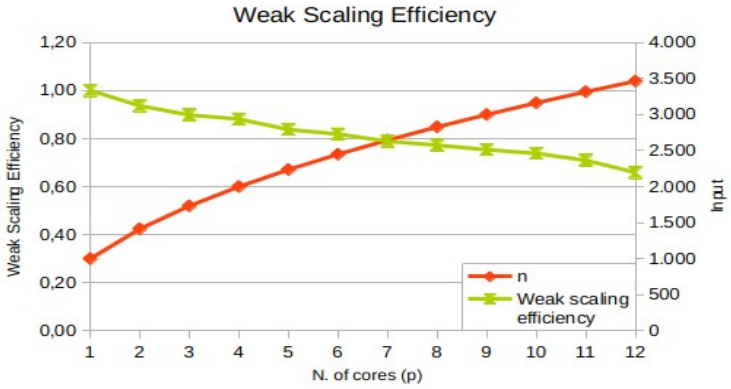


Grafico 3: Grafico WSE OpenMP

Versione MPI

Problemi riscontrati

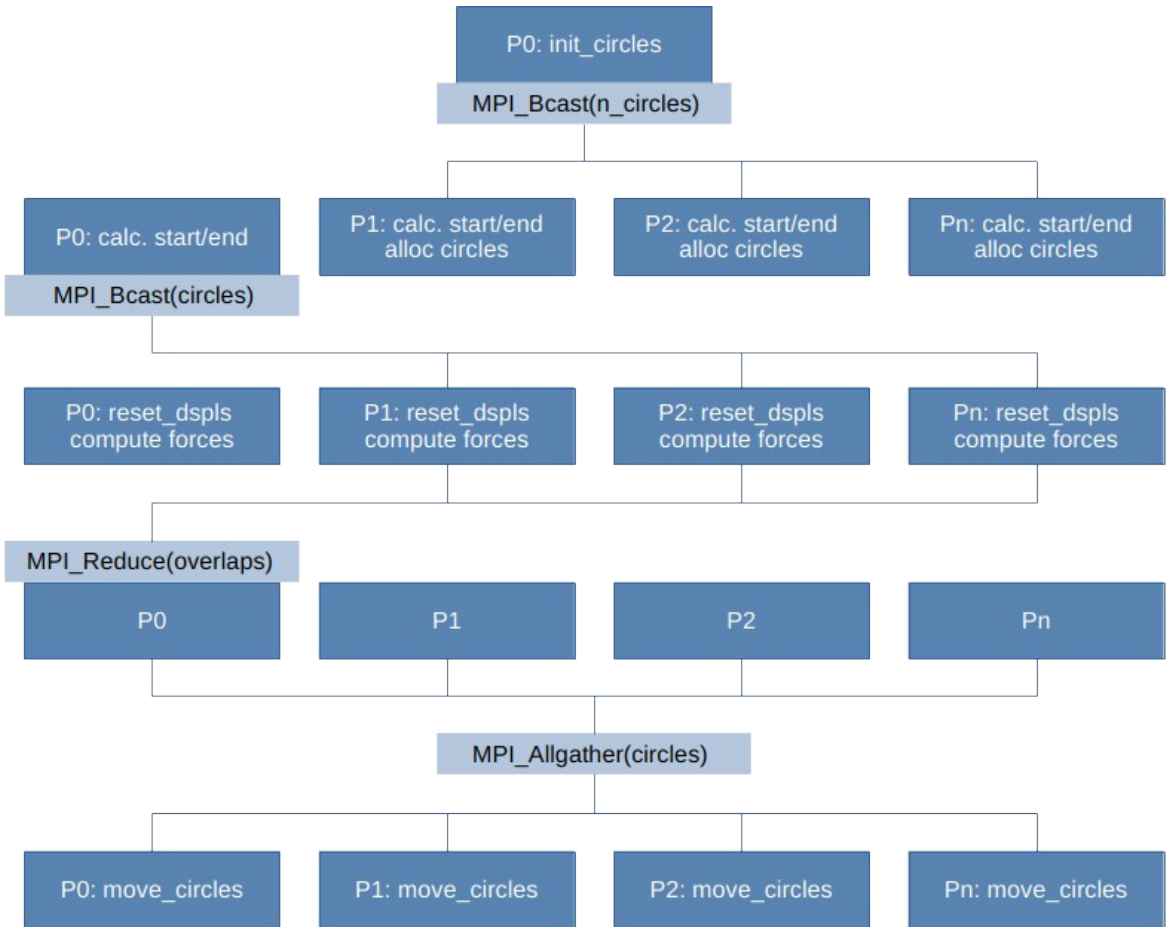
1. **Divisione del Lavoro**
   Ogni processo MPI deve gestire una porzione delle iterazioni sui cerchi per calcolare le forze tra tra tutte le coppie di cerchi.

   Ogni processo calcola i propri indici di partenza e fine dell'*array* di cerchi da elaborare, in modo da ottenere una distribuzione equa del carico. È stato necessario modificare leggermente le funzioni `compute_forces()`, `reset_displacements()`, `move_circles()` in modo che

accettassero come parametri gli indici di inizio e fine calcolati da ogni processo, per permetter l'accesso alle sezioni a loro assegnate.

2. **Distribuzione dei dati**
   Il processo 0 inizializza i dati dei cerchi, poi con una ***Broadcast*** si invia il numero di cerchi inizializzati a tutti i processi. A quel punto i processi allocano un *array* di cerchi locale, e con un'altra ***Broadcast*** l'*array* inizializzato dal processo 0 viene distribuito a tutti gli altri processi, che a questo punto avranno ognuno la propria copia locale dell'*array*.
3. **Calcolo delle Forze e Sovrapposizioni**
   Ogni processo calcola il numero di sovrapposizioni e le forze tra coppie di cerchi all'interno della porzione di *array* assegnatagli, le sovrapposizioni calcolate localmente da ogni processo vengono sommate tra loro attraverso una ***Reduce***, che aggrega i risultati calcolati da tutti i processi.
4. **Sincronizzazione e Movimento dei Cerchi**
   Dopo aver calcolato le forze, ogni processo calcola le nuove posizioni degli array, e con una ***Allgather*** si raccolgono gli scostamenti calcolati di tutti i cerchi, in modo che ogni processo conosca gli scostamenti calcolati da tutti gli altri processi e possa aggiornare le posizioni dei cerchi.

Schema delle comunicazioni



Analisi delle prestazioni

Le metriche utilizzate per misurare le prestazioni del programma MPI sono le stesse usate per la versione OpenMP. Si rimanda quindi alle formule (1),(2),(3),(4),(5), ritenendo valide le stesse (e necessarie) assunzioni e semplificazioni.

Le modalità di test del programma sono le medesime, ossia 5 prove per ogni dimensione del *pool* di processi, effettuate su 5000 cerchi e 100 iterazioni per il calcolo di *Speedup* e SSE, e 5 prove per ogni dimensione del *pool* di processi effettuate con un input base di 1000 cerchi e 100 iterazioni, moltiplicato di un fattore pari alla parte intera della radice quadrata del numero di processi impiegati, per il calcolo della WSE.

p	t1	t2	t3	t4	t5	Average WCT	Speedup	Strong Scaling Efficiency
1	61.33	61.33	61.31	61.31	61.31	61.32	1.00	1.00
2	32.25	30.69	30.69	30.71	30.69	31.00	1.98	0.99
3	20.50	20.50	20.50	20.50	20.51	20.50	2.99	1.00
4	15.39	15.39	15.39	15.39	15.39	15.39	3.98	1.00
5	12.32	12.32	12.32	12.38	12.33	12.34	4.97	0.99
6	10.29	10.29	10.28	10.28	10.29	10.29	5.96	0.99
7	8.85	8.84	8.84	8.84	8.84	8.84	6.94	0.99
8	7.75	8.87	7.75	7.75	7.75	7.97	7.69	0.96
9	6.90	6.91	6.89	6.92	6.92	6.91	8.87	0.99
10	6.28	6.21	6.23	6.21	6.21	6.23	9.85	0.98
11	5.81	5.73	5.73	5.94	5.72	5.79	10.59	0.96
12	7.04	6.45	6.61	6.52	6.55	6.64	9.24	0.77

Tabella 3: Calcolo Speedup e Strong Scaling Efficiency

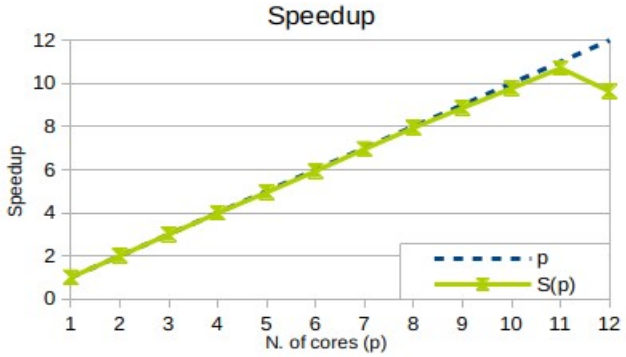


Grafico 4: Grafico Speedup MPI

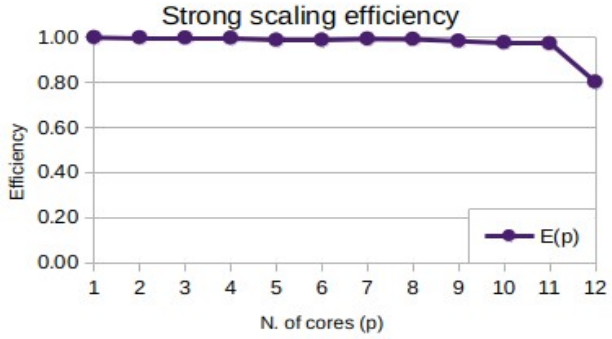


Grafico 5: Grafico SSE MPI

n	p	t1	t2	t3	t4	t5	Average WCT	Weak Scaling Efficiency
1,000	1	2.47	2.46	2.46	2.46	2.46	2.47	1.00
1,414	2	2.48	2.47	2.47	2.47	2.47	2.47	1.00
1,732	3	2.49	2.48	2.49	2.49	2.49	2.49	0.99
2,000	4	2.48	2.48	2.48	2.48	2.49	2.49	0.99
2,236	5	2.50	2.50	2.57	2.50	2.49	2.51	0.98
2,449	6	2.49	2.49	2.49	2.49	2.49	2.49	0.99
2,645	7	2.57	2.49	2.50	2.50	2.50	2.51	0.98
2,828	8	2.49	2.49	2.49	2.49	2.49	2.49	0.99
3,000	9	2.50	2.50	2.51	2.52	2.51	2.51	0.98
3,162	10	2.51	2.51	2.51	2.51	2.51	2.51	0.98
3,316	11	2.63	2.56	2.63	2.68	2.53	2.60	0.95
3,464	12	3.17	3.16	3.68	3.27	3.42	3.34	0.74

Tabella 4: Calcolo Weak Scaling Efficiency

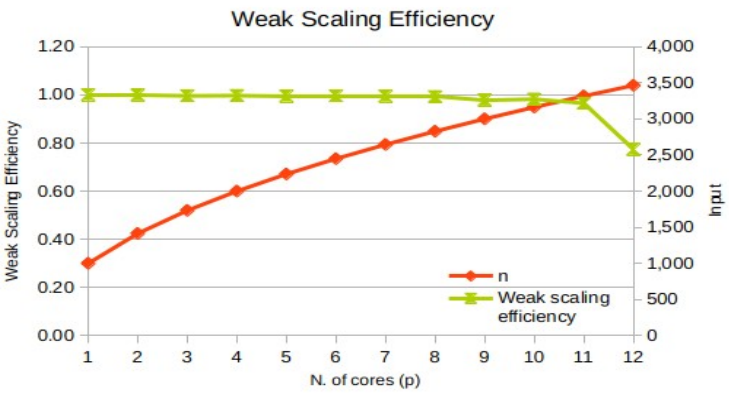


Grafico 6: Grafico WSE MPI

Osservazioni

Si osserva come l'andamento dello *Speedup* (Grafico 4) sia praticamente lineare, con SSE e WSE (Grafici 5 e 6) vicine a 1 utilizzando un numero di processi da 1 a 11, subendo un drastico calo con l'utilizzo di 12 processi.

La causa di tale comportamento è, con tutta probabilità, da ricondursi a interferenze di altri processi in esecuzione sul server di test.

Conclusioni

Dai dati raccolti emerge che MPI permette di scalare il programma in modo molto più efficiente rispetto a OpenMP. MPI offre uno *speedup* praticamente lineare e mantiene un'efficienza vicina al 100%, eccetto per le normali fluttuazioni dovute alla condivisione della CPU sul server utilizzato per i test. Questo risultato dimostra che MPI è altamente efficace nel gestire il calcolo parallelo distribuito, anche se richiede modifiche più significative al codice sorgente e l'implementazione di meccanismi di sincronizzazione e distribuzione dei dati non banali.

D'altra parte, OpenMP ha permesso di ottenere un buono *speedup*, sebbene con un'efficienza decrescente all'aumentare del numero di processi utilizzati. OpenMP si distingue per la sua facilità d'uso: permette di parallelizzare il codice mantenendo il programma praticamente invariato e utilizzando semplici direttive per la parallelizzazione, senza la necessità di distribuire i dati. Tuttavia, richiede attenzione per sincronizzare l'accesso alle porzioni di memoria condivisa, soprattutto per evitare race conditions.

Mettere a confronto le due tecnologie è complesso poiché nascono per scopi diversi e adottano logiche di funzionamento distinte. MPI è ideale per il calcolo distribuito su *cluster* di macchine, dove la comunicazione tra processi su nodi diversi è cruciale. OpenMP, invece, è più adatto per la parallelizzazione su sistemi condivisi, dove il focus è sulla facilità di implementazione e sulla gestione delle risorse condivise.

È interessante notare che, per un numero ridotto di processi paralleli, il programma MPI richiede circa il doppio del tempo rispetto alla versione OpenMP. Sebbene questa differenza non sia particolarmente significativa come metrica, diventa evidente che con l'aumentare del numero di processi, la maggiore efficienza e scalabilità di MPI superano questo svantaggio iniziale.

Non ho approfondito le ragioni specifiche di questo comportamento, ma ho ipotizzato che la causa principale sia l'*overhead* dovuto alle comunicazioni e alle modifiche implementate. Inoltre, la versione MPI necessita di un compilatore modificato, che potrebbe non applicare le stesse ottimizzazioni della versione del GCC disponibile sul server. Un altro fattore da considerare è che MPI richiede l'uso di un secondo eseguibile per l'esecuzione del binario, aggiungendo ulteriore *overhead*.

Possiamo concludere che, per applicazioni che richiedono un'elevata scalabilità e dove è possibile investire tempo e risorse per modificare il codice sorgente, MPI rappresenta la scelta migliore. Per situazioni dove si cerca una soluzione più rapida e meno invasiva per parallelizzare il codice su architetture *multi-core*, OpenMP rimane una valida alternativa, garantendo comunque significativi miglioramenti prestazionali.