Alma Mater Studiorum · Università di Bologna

Scuola di Scienze
Corso di Laurea Magistrale in Fisica

PARALLELIZZAZIONE E RINORMALIZZAZIONE NUMERICA

Relatore:
Prof. Fabio Ortolani

Presentata da:
Simone Vianello

Sessione II
Anno Accademico 2013/2014

Indice

Abstract

I processori multi core stanno cambiando lo sviluppo dei software in tutti i settori dell'informatica poiché offrono prestazioni più elevate con un consumo energetico più basso. Abbiamo quindi la possibilità di una computazione realmente parallela, distribuita tra i diversi core del processore. Uno standard per la programmazione multithreading è sicuramente OpenMP, il quale si propone di fornire direttive semplici e chiare per lo sviluppo di programmi su sistemi a memoria condivisa, fornendo un controllo completo sulla parallelizzazione.

Nella fisica moderna spesso vengono utilizzate simulazioni al computer di sistemi con alti livelli di complessità computazionale. Si ottimizzerà un software che utilizza l'algoritmo DMRG (Density Matrix Renormalization Group), un algoritmo che consente di studiare reticoli lineari di sistemi a molti corpi, al fine di renderlo più veloce nei calcoli cercando di sfruttare al meglio i core del processore.

Per fare ciò verrà utilizzata l'API OpenMP, che ci permetterà in modo poco invasivo di parallelizzare l'algoritmo rendendo così più veloce l'esecuzione su architetture multi core.

Capitolo 1

Introduzione

Il Density Matrix Renormalization Group (DMRG) è una tecnica numerica per lo studio dei sistemi a molti corpi. È stato sviluppato nel 1992 da Steven R. White presso l'Università della California a Irvine per superare i problemi che sorgono nell'applicazione del Numerical Renormalization Group (NRG) per i reticoli quantistici di sistemi a molti corpi. Da allora l'approccio è stato esteso ad una grande varietà di problemi in tutti i campi della fisica.

Il DMRG si applica a sistemi reticolari quantistici e consiste sostanzialmente in un troncamento dello spazio di Hilbert, mantenendo però un piccolo numero di stati rilevanti in modo da costruire le funzioni d'onda del sistema; per fare ciò usa i più probabili autostati di una matrice densità ridotta. Questa tecnica ha dimostrato di essere estremamente accurata anche

su grandi reticoli con migliaia di particelle e siti. L'efficienza e le potenzialità dell'algoritmo inoltre negli ultimi anni si sono evolute ulteriormente, in particolare sono stati sviluppati algoritmi per l'evoluzione temporale.

Data la natura iterativa del processo sono stati sviluppati software di simulazione in grado di gestire migliaia di siti. Il calcolo effettivo del DMRG però, vista la complessità dei calcoli e la grandezza delle matrici usate, viene eseguito in tempi mediamente lunghi; ed è proprio questo problema dei tempi di esecuzione delle simulazioni che è stato affrontato. Infatti quello che si è provato a fare è stata un'ottimizzazione di un programma di simulazione del DMRG, sviluppato dal Dipartimento di Fisica dell'Università di Bologna e in particolar modo dal professor Fabio Ortolani, al fine di renderlo più veloce nei calcoli usando delle tecniche di parallelizzazione.

Per fare ciò abbiamo utilizzato OpenMP, un API per sistemi a memoria condivisa di programmazione parallela, che comprende direttive per i compilatori, funzioni di libreria, e variabili di ambiente che possono essere usate per scrivere programmi in vari linguaggi, come il C/C++ e il Fortran.

Capitolo 2

Density-Matrix Renormalization Group

Il gruppo di rinormalizzazione della matrice densità (DMRG) è un efficace metodo utilizzato per descrivere la fisica delle basse energie di sistemi a molti corpi con elevata precisione.

Uno dei problemi principali della fisica quantistica dei sistemi a molti corpi è che lo spazio di Hilbert cresce esponenzialmente con il numero delle particelle. Per esempio, se consideriamo una catena lunga L di spin $S=1/2,$ questa avrà 2^L gradi di libertà. Il DMRG è una tecnica iterativa che consente di abbassare i gradi di libertà del sistema tenendo conto solo di quelli effettivi per lo studio dello stato in esame.

L'algoritmo consiste nel dividere il sistema in due blocchi (non è necessario

che siano delle stesse dimensioni) divisi da due siti. L'insieme dei due blocchi più i due siti centrali è noto come super blocco. In questo modo viene creata una versione ridotta del sistema iniziale, della quale è possibile farne una trattazione più semplice rispetto a quella di partenza. Così facendo si ha però una scarsa precisione, ma essendo un metodo iterativo la precisione può essere migliorata.

Successivamente lo stato del super blocco trovato va proiettato sui due blocchi mediante la matrice densità, selezionando così gli stati più rilevanti. Ora viene fatto crescere di un sito il blocco di sinistra a discapito di quello di destra creando così un nuovo super blocco. Il procedimento descritto può essere iterato per aumentare la precisione del calcolo.

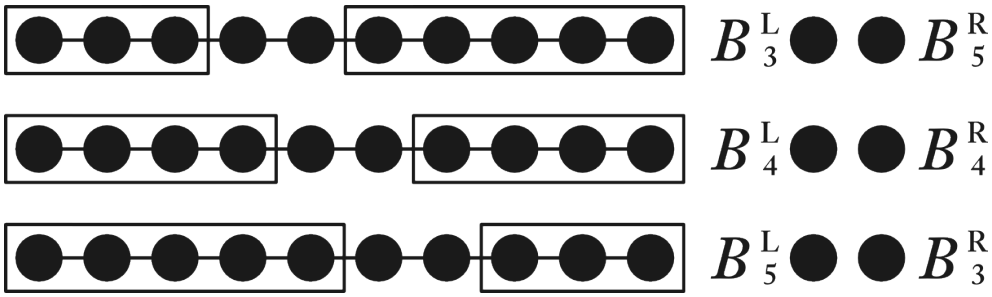

Figura 2.1: Creazione del sistema in blocchi sinistro e destro, secondo DMRG.

2.1 Metodo dei super blocchi

Il metodo è abbastanza generale e può essere applicato a vari modelli unidimensionali tipo il modello di Ising o ad una catena di spin di Heisenberg. Lo scopo che si prefigge è quello di diagonalizzare un super blocco composto da tre o più blocchi più piccoli e con questi stati ottenere un'approssimazione degli stati di uno dei sotto blocchi.

Considerando per esempio una catena di spin di Heisenberg si immagini di dividerla in 4 catene (blocchi) più piccole, l'unione di queste catene da il super blocco. Infine le funzioni d'onda del super blocco vengono proiettate su un blocco più piccolo e vengono tenuti questi stati come se fossero gli stati del blocco. Il sistema funziona molto bene per i sistemi di singola particella, in quanto la proiezione ha un unico valore e aumenta di precisione con l'aumentare dei blocchi considerati.

Nel caso di funzioni d'onda a molte particelle la proiezione non ha un unico valore, ma tuttavia alcuni di questi valori non sono fisicamente rilevanti e la matrice densità ci aiuta a capire quali di questi stati sono rilevanti per lo studio del sistema in esame.

2.1.1 Matrice di densità ridotta

Come suggerisce il nome, l'algoritmo DMRG non opera su stati quantistici puri, ma su matrici densità. In generale, si può pensare ad un sistema una matrice densità ρ come una miscela statistica di stati quantistici puri $|\psi_i\rangle$:

$$\rho = \sum_i p_i |\psi_i\rangle\langle\psi_i|, \tag{2.1}$$

dove p_i può essere interpretato come la probabilità di trovare il sistema nello stato $|\psi_i\rangle$, inoltre la matrice densità ha due importanti proprietà: ha la traccia unitaria ed è semidefinita positiva. Queste matrici svolgono un ruolo importante quando si è interessati allo studio di un piccolo sottosistema incorporato in un ambiente più grande; si consideri un sistema composto dai sottosistemi A e B, questo può essere rappresentato da:

$$|\psi\rangle = \sum_{ij} c_{ij} |i\rangle_A |j\rangle_B \equiv \sum_{ij} c_{ij} |i\,j\rangle . \tag{2.2}$$

Possiamo quindi definire la matrice densità ridotta di A come la traccia parziale su B:

$$\rho_A = Tr_B\{\rho\} = \sum_{ii'} \sum_{ij} \langle i\,j|\rho|i'\,j\rangle |i\rangle\langle i'|. \tag{2.3}$$

2.2 DMRG

Questo approccio prevede che venga preso in considerazione un sistema più grande che comprenda quello in esame, quindi per trovare la matrice densità si dovrà diagonalizzare l'Hamiltoniana di un sistema più grande (super blocco); successivamente gli autostati di questo super blocco saranno utilizzati al fine di trovare la matrice densità per lo stato del nostro sistema. L'algoritmo dipende principalmente quindi dalla costruzione del super blocco e dalla scelta dei sui autostati, che saranno poi utilizzati per la costruzione della matrice densità.

Gli algoritmi più efficienti utilizzano solo un autostato del super blocco per costruire la matrice densità del blocco. Un delle caratteristiche più importanti di questo algoritmo è che la precisione sale con l'aumentare degli stati, si è visto infatti che cresce esponenzialmente con il numero di essi.

Il concetto chiave dell'algoritmo è pensare al sistema a molti corpi in esame come se fosse composto da un sistema S di dimensioni L collegato a un ambiente esterno. Supponiamo di avere un set di N stati $\{|\phi_S\rangle\}$ che ci consente di fornire una buona approssimazione delle proprietà fisiche del sistema S, possiamo dunque aumentarle a $L+1$ con il seguente procedimento:

1. Formare il nuovo sistema S' di dimensione $L+1$ combinando gli stati che descrivono S con l'intero spazio di Hilbert che descrive il sito aggiuntivo.

2. Costruire un super blocco di dimensione $2L+2$ combinando il sistema allargato S' con l'ambiente esterno E:

3. Trovare lo stato fondamentale $|\psi\rangle$ dell'Hamiltoniana del super blocco diagonalizzandola.

4. Calcolare la matrice densità ridotta per S': $\rho_{S'}=Tr_E\{|\psi\rangle\langle\psi|\}$.

5. Diagonalizzare la matrice densità ridotta e mantenere solo gli autovettori corrispondenti agli N più grandi autovalori.

6. Esprimere tutti gli operatori necessari per descrivere il sistema S' in questa nuova base.

7. Continuare questa iterazione fino a che non si ottiene una convergenza sufficiente con gli osservabili (ad esempio, l'energia stato fondamentale).

In pratica, si può spesso ottenere risultati significativamente migliori tenendo conto degli effetti dovuti alle dimensioni finite. Questo può essere fatto in un modo relativamente semplice tenendo traccia del sistema e l'ambiente separatamente. Una volta che questo algoritmo porta il sistema a raggiungere le dimensioni desiderate, si può successivamente aumentare il sistema a scapito dell'ambiente. Quando l'ambiente raggiunge una dimensione minima, la procedura è invertita e l'ambiente viene aumentato a spese del sistema. Più operazioni di questo tipo possono essere eseguite fino

a quando l'energia dello stato fondamentale converge ad un valore migliore. Durante ogni step possiamo utilizzare anche la stima per lo stato fondamentale ottenuto nella fase precedente come stato iniziale della nostra procedura di diagonalizzazione, il che porterà ad un notevole miglioramento dei risultati. In generale, l'errore ε del procedimento DMRG è dato da un errore di troncamento, che è dovuto alla somma di tutte le approssimazioni compiute nei vari step. I valori tipici sono $\varepsilon \sim 10^{-10}$ per $N \sim 100$.

2.2.1 Catena infinita

Il super blocco iniziale è composto da 4 siti: $B_1^L \bullet \bullet B_1^R$, dove B_1^L e B_1^R contengono solo uno stato. Successivamente viene diagonalizzato e calcolata la matrice densità. Nello step successivo si prende in esame lo stato $B_2^L = B_1^L \bullet$ e viene così creato il super blocco $B_2^L \bullet \bullet B_2^R$, dove $B_2^L = B_2^R$ viene creato per riflessione. Si continua poi così l'algoritmo per gli step successivi: $B_{i+1}^L = B_i^L \bullet$.

A ciascun step dell'algoritmo entrambi i blocchi aumentano di un sito mentre la lunghezza effettiva della catena cresce di due, spingendo sempre più lontano le estremità della catena dai siti centrali. Così facendo dopo molte iterazioni ciascun blocco rappresenterà metà di una catena infinita.

I due siti centrali servono per rendere più accurato l'algoritmo, in quanto B_i^R è rappresentato da un Hamiltonia approssimata, mentre i siti al centro si

possono rappresentare esattamente. Inoltre il numero degli autovalori non nulli della matrice densità deve essere più grande degli stati conservati, tranne nei primi passaggi dove tutti gli stati del blocco devono esserlo.

2.2.2 Catena finita

Prendiamo ora in esame una catena composta da L elementi, possiamo procedere come per il caso infinito per $L/2-1$ iterazioni, finché il super blocco usato non arriva alle dimensioni di L. Una volta arrivati alla configurazione $B^L_{L/2-1}\bullet\bullet B^R_{L/2-1}$ si procederà aggiungendo un sito al blocco di destra e togliendolo a quello di sinistra, $B^L_{L/2}\bullet\bullet B^R_{L/2-2}$, da questo momento in poi ovviamente il sistema conterrà L siti. Si dovrà procedere in questo modo iterando il processo finché non si arriverà alla configurazione $B^L_1\bullet\bullet B^R_{L-3}$, solitamente poi si procede fino a tornare alla configurazione $B^L_{L/2-1}\bullet\bullet B^R_{L/2-1}$. Da notare che durante queste serie di iterazioni alcune matrici densità possono essere riutilizzate.

2.3 Catena di spin di Heisenberg

Un sistema quantistico di spin è una collezione di particelle vincolate ad occupare in modo permanente i siti di un reticolo. Tali particelle interagiscono a distanza per accoppiamento dei rispettivi spin vicendevolmente e o con un campo esterno. Ciò comporta un'evoluzione temporale, per cui le orientazioni degli spin cambiano nel tempo in accordo alle leggi fisiche che definiscono il sistema.

L'Hamitoniana del modello di Heisenberg, che descrive un sistema di interazione spin-spin (adiacenti) , in assenza di campo magnetico esterno è data da:

$$\boldsymbol{H} = -J \sum_{kj} \vec{S}_k \vec{S}_j, \tag{2.4}$$

si può notare che è simmetrica rispetto trasformazioni di $SU(2)$, questo implica che gli spin totali saranno conservati e rappresenteranno una costante del moto.

Consideriamo ora il caso unidimensionale di una catena di spin $S = 1/2$ composta da N siti e aggiungiamo condizione al bordo periodiche, ossia la particella al posto j e quella al posto $N+j$ coincidono. In questo modo ogni spin interagirà con i due vicini e la (2.4), riscritta in termini delle componenti, diventa:

$$\boldsymbol{H} = -\sum_{j=1}^{N} \left(J_x S_j^x S_{j+1}^x + J_y S_j^y S_{j+1}^y + J_z S_j^z S_{j+1}^z \right), \qquad (2.5$$

le costanti di accoppiamento J^α con $\alpha = x, y, z$ sono state considerate indipendenti. In questo modo la simmetria su $SU(2)$ viene persa. I casi particolari sono $J_x = j_y \neq J_z$ e $J_x = j_y = J_z$ che prendono il nome rispettivamente di catena XXZ e catena XXX, il caso invece con $J_z = 0$ e chiamato catena XY.

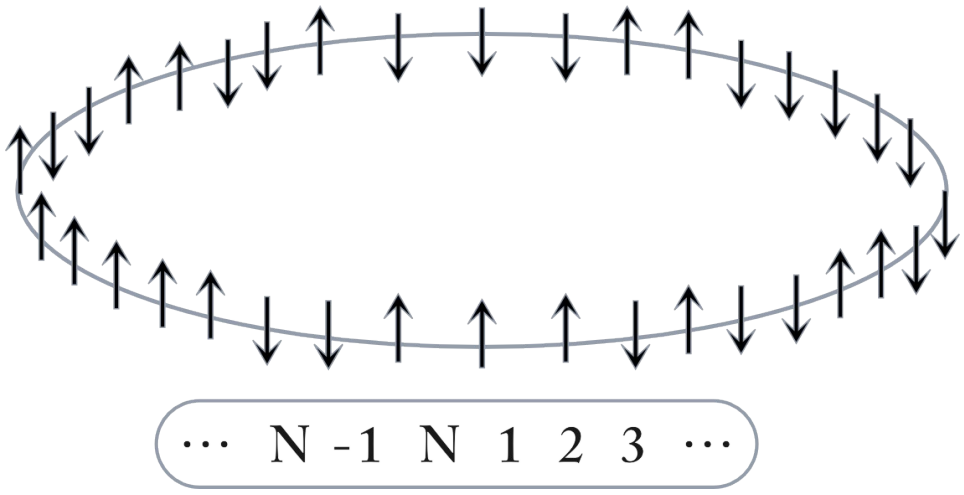

Figura 2.2: Catena di N spin con condizioni a bordo periodiche.

2.3.1 Applicazione del DMRG alla catena di Heisenberg

Vogliamo ora dare l'esempio di una semplice applicazione dell'algoritmo di una catena di spin di Heisenberg infinita con $S = 1$. Si parte a simulare la

catena da 4 siti: il primo sito corrisponde a B^L; l'ultimo B^R corrisponde all'ambiente esterno e i due centrali, $\bullet^1\bullet^2$, sono i siti aggiunti.

$$
\begin{aligned}
&B^L \rightarrow \mathcal{H}_L\{|u_i\rangle\}, \boldsymbol{H_L}, S_{xL}, S_{yL}, S_{zL},\\
&\bullet^1 \rightarrow \mathcal{H}_1\{|t_i\rangle\}, S_{x1}, S_{y1}, S_{z1},\\
&\bullet^2 \rightarrow \mathcal{H}_2\{|s_i\rangle\}, S_{x2}, S_{y2}, S_{z2},\\
&B^R \rightarrow \mathcal{H}_R\{|r_i\rangle\}, \boldsymbol{H_R}, S_{xR}, S_{yR}, S_{zR}.
\end{aligned}
\tag{2.6}
$$

Al momento iniziale tutti e quattro gli spazi di Hilbert sono equivalenti, tutti gli operatori di spin sono equivalenti e $\boldsymbol{H_L}=\boldsymbol{H_R}=0$.

Step 1: Creazione dell'Hamiltoniana del super blocco

Lo spazio di Hilbert del super blocco e l'Hamiltonia (si è posto $J=1$ per comodità) possono essere così costruiti:

$$
\begin{aligned}
\mathcal{H}_{SB} &= \mathcal{H}_L \otimes \mathcal{H}_1 \otimes \mathcal{H}_2 \otimes \mathcal{H}_R,\\
|f\rangle &= |u\rangle \otimes |t\rangle \otimes |s\rangle \otimes |r\rangle,\\
\boldsymbol{H_{SB}} &= \boldsymbol{H_L} + \boldsymbol{H_R} + \sum_{ij} \left(S_i^x S_j^x + S_i^y S_j^y + S_i^z S_j^z\right).
\end{aligned}
\tag{2.7}
$$

L'operatore $\boldsymbol{H_{SB}}$ è rappresentato da una matrice di dimensione $(d*3*3*d)\times(d*3*3*d)$, dove $d=\dim(\mathcal{H}_L)\equiv\dim(\mathcal{H}_R)$.

Step 2: Diagonalizzazione della nuova Hamiltoniana

Ora è necessario scegliere un autostato dell'Hamiltoniana per cui degli osservabili sono stati calcolati, questo è il target state; da notare che all'inizio è possibile scegliere lo stato fondamentale. Questo step è quello che richiede più tempo all'algoritmo.

Step 3: Riduzione della matrice densità

Calcolare la matrice di densità ridotta $\rho_{B_2^L}$, di dimensione $(d*3)\times(d*3)$, per il blocco iniziale $B_2^L = B^L \bullet^1$ a partire dalla matrice ρ del super blocco. Successivamente diagonalizzare la matrice ottenuta e costruire la matrice T di dimensione $m\times(d*3)$, le cui righe sono gli m autovettori associati agli m più grandi autovalori e_α di $\rho_{B_2^L}$. In questo modo la matrice ottenuta è formata dagli autostati più di $\rho_{B_2^L}$. Il numero di autostati da conservare deve essere scelto in modo che si verifichi:

$$1-\sum_{\alpha=1}^{m} e_\alpha \approx 0. \qquad (2.8)$$

Step 4: Nuovi operatori e blocchi

Creare le $(d*3)\times(d*3)$ matrici degli operatori del blocco B_2^L e B_2^R:

$$\begin{aligned} \boldsymbol{H}_{\boldsymbol{B^L\bullet^1}} &= \boldsymbol{H_L}\otimes\mathbb{I}+S_{xL}\otimes S_{x1}+S_{yL}\otimes S_{y1}+S_{zL}\otimes S_{z1}, \\ \boldsymbol{H}_{\boldsymbol{\bullet^2 B^R}} &= \boldsymbol{H_R}\otimes\mathbb{I}+S_{xR}\otimes S_{x2}+S_{yR}\otimes S_{y2}+S_{zR}\otimes S_{z2}. \end{aligned} \tag{2.9}$$

Infine creare le matrici $m\times m$ rappresentative degli operatori dei nuovi blocchi B^L e B^R facendo un cambiamento di base con la trasformazione T:

$$\begin{aligned} \boldsymbol{H^L} &= T\,\boldsymbol{H}_{\boldsymbol{B^L\bullet^1}}T^\dagger, \\ \boldsymbol{H^R} &= T\,\boldsymbol{H}_{\boldsymbol{\bullet^2 B^R}}T^\dagger. \end{aligned} \tag{2.10}$$

Il procedimento deve essere svolto analogamente anche per le matrici di spin. A questo punto l'iterazione è conclusa e l'algoritmo riprende dallo step 1. il procedimento verrà stoppato solo quando si raggiungeranno le dimensione volute della catena.

Capitolo 3

Calcolo parallelo

Il calcolo parallelo consiste nel suddividere e svolgere simultaneamente un codice sorgente su più microprocessori o più core, aumentandone così la velocità di esecuzione rispetto a quella che si avrebbe utilizzando un unico processore. Il principio base sta nella suddivisione del problema in sotto problemi che poi verranno eseguiti nello stesso istante spartendo così il carico di lavoro.

L'algoritmo inizialmente è concepito come un flusso seriale di istruzioni che vengono processate una alla volta, solo quando una precedente istruzione è stata eseguita si passerà a quella successiva. Il calcolo parallelo, invece, spezza il problema in parti indipendenti fra loro eseguendo così più parti dell'algoritmo simultaneamente. Comunque sia non si avranno prestazioni

proporzionali al numero di processori presenti nel sistema in quanto molti algoritmi resteranno sequenziali.

Questo tipo di approccio però porta con se dei problemi in quanto necessita di algoritmi appositi, perciò non è detto che un codice che funziona correttamente per un singolo processore funzioni altrettanto bene se viene ottimizzato in questo modo, per cui, se si decide di intraprendere questo percorso uno dei problemi più grossi potrebbe essere quello della sincronizzazione, cioè più processi potrebbero voler accedere allo stesso indirizzo di memoria nello stesso istante, questo porterebbe le varie parti di codice ad interferire fra loro.

3.1 Programmazione parallela

La questione fondamentale in questo approccio è l'individuazione di segmenti di codice indipendenti, cioè le variabili utilizzate durante l'esecuzione e quelle in uscita non devono essere in comune fra le parti svolte in parallelo. Le condizioni di Bernstein stabiliscono quando due segmenti di codice possono essere eseguiti contemporaneamente:

$$\begin{aligned} I_2 \cap O_1 &= \emptyset , \\ I_1 \cap O_2 &= \emptyset , \\ O_1 \cap O_2 &= \emptyset . \end{aligned} \qquad (3.1)$$

dove I_1 e I_2 rappresentano le variabili in ingresso rispettivamente del primo e del secondo processo, mentre O_1 e O_2 quelle in uscita. La prima e seconda condizione implicano che le variabili in entrata di entrambi i segmenti non devono dipendere da operazioni svolte nell'altro segmento, introducendo così una dipendenza di flusso; mentre la terza condizione implica che le variabili in uscita non devono essere le stesse, in quanto il risultato finale sarebbe solamente quello scritto per ultimo. Queste condizioni dunque non consentono la condivisione di memoria fra i processi da parallelizzare, quindi sono necessari degli strumenti di sincronizzazione.

I segmenti di codice che vengono eseguiti in parallelo sono chiamati threads, solitamente condividono risorse tra loro, infatti più threads possono accedere ad una stessa variabile e modificarne il contenuto, quest'accesso però non può avvenire simultaneamente in quanto le variabili sono memorizzate nella RAM, ma può succedere che un altro thread invece abbia

bisogno del vecchio valore associato alla variabile:

Thread A	**Thread B**
Legge variabile X	*Legge variabile X*
Modifica X	*Modifica X*
Riscrive X	*Riscrive X*

Se il thread B legge la variabile X dopo la modifica da parte del thread A o viceversa, il programma darà risultati non corretti. In questo caso sarà necessario utilizzare un blocco che prevenga la modifica della variabile da parte di altri threads. Si creerà così una regione critica all'interno della quale i threads opereranno sulla stessa variabile senza interferire tra loro:

Thread A	**Thread B**
Blocca variabile X	*Blocca variabile X*
Legge X	*Legge X*
Modifica X	*Modifica X*
Riscrive X	*Riscrive X*
Sb*locca X*	Sb*locca X*

In questo modo un thread dovrebbe essere indipendente dal resto del programma. In altri casi invece potrebbe essere richiesto che alcuni threads vengano eseguiti in sincronia, sarà dunque necessario l'uso di barriere che aiutino la sincronizzazione tra i vari processi. Questo però, non è sempre facile in quanto la sincronizzazione dei threads dipende dall'ambiente in cui il programma viene lanciato.

Ovviamente il codice scritto in questo modo deve produrre risultati come se fosse scritto in forma sequenziale. In altre parole un codice è coerente se i risultati ottenuti da ogni sua singola esecuzione sono gli stessi che si sarebbero ottenuti facendo tutte le operazioni in ordine sequenziale.

3.1.1 Hardware

Esistono diverse tipologie di hardware nelle quali è possibile eseguire un calcolo distribuito, la principale differenza sta nella gestione della memoria. Infatti i vari processori possono condividere la stessa memoria oppure dividerla in modo che ogni processore abbia la propria. Infine si possono combinare le due tipologie precedenti in modo da avere una parte condivisa e una propria, tenendo conto però che l'accesso alla memoria propria sarà sempre più veloce di quello alla memoria condivisa.

I processori fanno uso della memoria cache, una memoria molto veloce e vicina al processore, la quale solitamente mantiene i dati ai quali si accede più frequentemente dalla memoria principale, rendendo così il loro accesso notevolmente più veloce , in quanto i tempi di latenza saranno minori. I sistemi informatici paralleli però hanno una notevole difficoltà a gestirla, difatti, può memorizzare più volte lo stesso valore in posizioni di memoria diverse e questo porterebbe ad una esecuzione errata de codice. Bisogna quindi richiedere una coerenza della cache in modo che riesca a tenere traccia dei dati memorizzati ed eliminare eventuali copie.

Si possono dunque individuare diverse tipologie di sistemi adatte all'uso di algoritmi paralleli che differiscono sostanzialmente dall'hardware, i più comuni possono essere:

- Processori Multicore: sono sostanzialmente CPU composte da due o più core montati assieme; questa architettura permette di aumentare la velocità di calcolo senza aumentare la frequenza di clock diminuendo così il calore dissipato dal sistema.

- Sistema Multiprocessore Simmetrico: è un sistema composto da più processori che condividono un unica memoria centrale, questi processori, che hanno una loro propria memoria centrale (chache memory), lavorano autonomamente elaborando ciascuno processi differenti ma con la possibilità di condividere risorse.

- Calcolo Distribuito: è costituito da un insieme di computer autonomi che interagiscono tra loro comunicando attraverso una rete.

- Coumputer Cluster: è un insieme di computer legati mediante una rete in modo da sembrare sotto alcuni aspetti un unico computer.

3.1.3 Software

A seconda del tipo di architettura (memoria condivisa o memoria distribuita) sono state sviluppate diverse API (Application Programming Interface) e librerie per la programmazione parallela. I linguaggi di programmazione per la memoria condivisa comunicano manipolando le variabili memorizzate, quelli per la memoria distribuita invece lo fanno scambiandosi messaggi. OpenMP (Open Multiprocessing) è una delle API più diffuse per la programmazione di sistemi a memoria condivisa, mentre MPI (Message Passing Interface) è quella più diffusa per i sistemi a memoria condivisa.

Principalmente possiamo avere due tipi di approcci alla parallelizzazione di un codice:

- Parallelismo a livello di istruzione: consiste nel dividere e riordinare in gruppi le istruzioni per farle poi eseguire in parallelo dal processore. Questo metodo può essere effettuato anche solo a livello hardware, infatti molti processori moderni sono in grado di ricercare ed eseguire pezzi di codice in parallelo.

- Parallelismo a task: consiste nel dividere un algoritmo in sotto algoritmi, i quali vengono poi eseguiti contemporaneamente e in modo cooperativo dal processore. Questi sotto algoritmi possono operare sia sulle stesse variabili che averne di indipendenti.

Esiste comunque un limite teorico all'aumento della velocità di un programma che è legata principalmente all'algoritmo e non al numero dei processori, infatti prima o poi si arriverà ad un punto in cui l'algoritmo non potrà più essere parallelizzato ulteriormente. Continuando ad aumentare il numero di processori si arriverà dunque ad un punto in cui non ci sarà più possibile aumentare la velocità di esecuzione del programma. Questo principio è dato dalla legge di Amdahl, la quale, può essere espressa sinteticamente con "*make the common case fast*".

3.2 OpenMP

Si tratta di un API multi-piattaforma che consente la creazione di programmi su architetture a memoria condivisa. Le sue librerie sono supportate sia da C/C++ (le funzioni implementate sono incluse in omp.h) che da Fortran e può essere utilizzato in ambienti diversi come per esempio Linux, Mac OS X e Windows. È gestito da un consorzio no profit OpenMP ARB al quale partecipano anche vari produttori di hardware e software, tra cui AMD, IBM,Intel, Microsoft Corporation e Oracle.

OpenMP è basato sul concetto di multithreading, nel quale un master thread crea dei sotto threads i quali vengono eseguiti simultaneamente e il sistema li assegna ai processori disponibili in base a vari fattori, come per esempio il carico di lavoro del computer. Al livello di codice i threads vengono assegnati con una apposita direttiva che li crea prima dell'esecuzione del programma e ad ognuno è assegnato un id che li identifica (il master thread è lo 0). Durante l'esecuzione del programma, una volta raggiunta una zona del codice che deve essere svolta in parallelo, il master thread passa il controllo ai sotto threads, si stoppa, attende che finiscano il loro lavoro e una volta finito riprenderà il controllo fino al termine o fino alla prossima zona parallelizzata. In questo modo ogni thread eseguirà in modo indipendente il codice ad esso assegnato.

Sostanzialmente l'API di OpenMP permette di gestire le seguenti operazioni: creare e gestire threads, assegnare e distribuire il lavoro tra i threads, specificare quali variabili sono private e quali sono condivise tra i

threads e infine coordinare l'accesso alle variabili condivise; quindi potrebbe sembrare una buona scelta per tentare una parallelizzazione dell'algoritmo DMRG, infatti tra i sui vantaggi c'è sicuramente la semplicità d'uso e il fatto che le variabili sono completamente gestite dalle sue direttive ma sopratutto non sarà necessario apportare modifiche drastiche al codice in quanto basterà intervenire solo nei punti in cui si necessita una parallelizzazione. Nonostante ciò l'utilizzo presenta ancora diversi problemi, primo fra tutti la possibilità di scrivere falsa condivisione del codice o che manchi una gestione degli errori affidabile, rendendo così il debug molto complicato.

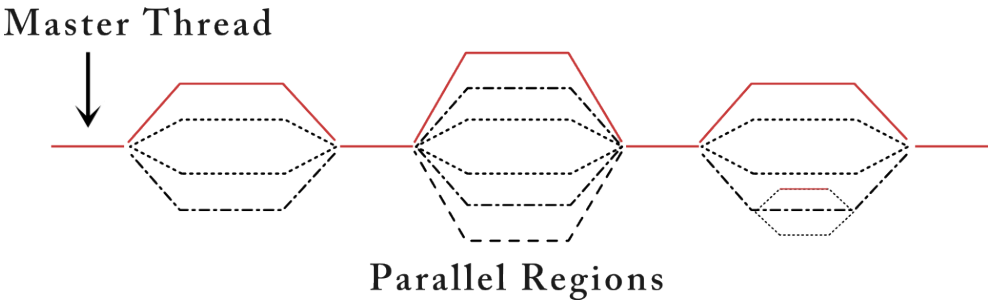

Figura 3.1: Schema threads con OpenMP

3.2.1 Gestione della memoria

I punti fondamentali del modello di memoria sono:

- I singoli thread "collaborano" attraverso la memoria condivisa, la quale può essere letta/scritta da tutti.

- I threads hanno spazio per variabili proprie accessibili in modalità esclusiva e memorizzate nello stack del thread.

- Le variabili originali sono quelle definite nel programma seriale. Per esempio una variabile di nome `x` è condivisa se per ogni thread il nome `x` si riferisce alla stessa variabile originale. Invece la variabile di nome `x` e' privata se ne viene creata una nuova istanza nello stack dei threads.

- OpenMP fornisce solo gli strumenti per coordinare i threads, ma la corretta parallelizzazione è compito del programmatore.

Le variabili possono essere dichiarate, all'inizio di una regione critica: `shared` o `private`.

Dichiarare una variabile `shared` vuol dire che ad essa potranno accedere tutti i threads e quindi potranno modificarla. Si può usare una variabile così definita quando occorre scambiare valori con altri threads. Questo sistema però non consente sincronizzazione, è garantito che solo un thread per volta modifichi tale valore.

Dichiarare una variabile `private` vuol dire che ogni thread avrà la sua copia locale e potrà servire per calcoli parziali. Occorre inizializzare sempre una variabile privata. Si può evitare questo usando la clausola `firstprivate` che inizializza tale variabile al valore della variabile originale. Per recuperare il valore di una variabile privata dall'ultimo thread

eseguito ed assegnarlo a quella originale si usa la clausola `lastprivate`. Di default tutte le variabili sono `shared`.

3.2.2 Prodotto tra matrici

Analizzando il codice si può notare che nel file action.cc la parte più pensante e lenta del calcolo consiste nella moltiplicazione di grossi blocchi di matrice dell'ordine di diverse centinaia di righe e colonne. Quindi si è pensato di partire da lì ed utilizzare le direttive della libreria per cercare di rendere il tutto più veloce.

Un semplice algoritmo per la moltiplicazione di matrici in C può essere scritto mediante l'annidamento di tre cicli `for:`

```
for (long k=0; k<intsz; k++)
 for (long j=0; j<rgtsz; j++)
  for (long i=0; i<lftsz; i++)
   p[i+j*lftsz]+=a[i+k*lftsz]*b[k+j*intsz];
```

Codice 3.1: Algoritmo moltiplicazione matrici (lftsz x intsz) x (intsz x rgtsz).

Nella libreria sono presenti delle direttive per parallelizzare dei cicli `for` in maniera molto semplice e diretta, infatti si possono aprire in linea di principio dei threads per suddividere il carico di lavoro del ciclo e

suddividerlo in tanti sotto cicli che poi verranno effettuati da processori diversi.

La direttiva `#pragma omp parallel for` specifica che le iterazioni del ciclo contenuto al suo interno devono essere distribuite fra i vari threads; in questo modo si parallelizza solo il ciclo `for` più esterno, mentre i cicli interni saranno eseguiti singolarmente da ciascun thread. Esiste anche la possibilità di utilizzare la clausola `collapse(n)`, che ha l'effetto di far collassare più cicli `for` innestati in un unico ciclo, specificando gli `n` cicli consecutivi da accorpare (in questo caso andrebbe richiamata la `collapse(2)`).

Tuttavia si è rivelata inefficace, in quanto, anche se l'esecuzione risultava più veloce si poteva riscontrare una differenza nei risultati rispetto ad una esecuzione del codice in maniera sequenziale che non poteva essere trascurata. Altri tentativi per cercare di parallelizzare quell'algoritmo hanno portato tutti allo stesso esito fallimentare o ad un'esecuzione del programma addirittura più lenta della versione sequenziale.

I risultati indicavano una cattiva gestione della cache da parte della libreria, anche se sembrava poco credibile visto che il primo tentativo è stato fatto con una delle direttive più usate. Dopo vari test è stato visto che OpenMP aveva problemi a moltiplicare in questo modo matrici che erano state memorizzate in un unico vettore, quindi si è provato a modificare l'algoritmo rendendo nuovamente le matrici bidimensionali ma questo ha portato ad un aumento spropositato dei tempi. Ogni tentativo si rivelato

dunque infruttuoso e dai risultati errati si è supposto che il problema principale fosse la grandezza delle variabili e una cattiva gestione della cache da parte della libreria.

Questo primo approccio ha fatto pensare che non fosse possibile un'effettiva parallelizzazione degli algoritmi che riguardavano la moltiplicazione tra matrici, i risultati più discordanti si sono ottenuti mentre si cercava di cambiare e velocizzare un algoritmo atto alla moltiplicazione di matrici sparse (matrici con molti zeri). Quindi dopo numerosi tentativi si abbandonò questa strada e si provò a vedere l'algoritmo sotto un altro punto di vista.

3.2.3 Tasks

OpenMp offre la possibilità di creare dei tasks, ovvero dei pezzi di codice che verranno gestiti come veri e propri processi indipendenti dal sistema. Possono essere visti come unità indipendenti di lavoro con proprie variabili interne ed è il sistema a decidere quando eseguirli; per questo la loro gestione necessita di barriere che blocchino il programma finché tutti i tasks non avranno svolto il loro compito.

Si è pensato dunque di tralasciare l'algoritmo che moltiplicava le matrici e si è notato che era possibile moltiplicarle contemporaneamente. Infatti nonostante ci fossero alcuni problemi, sempre dovuti all'approssimazione dei risultati finali, creando dei tasks indipendenti si sono potute effettuare più moltiplicazioni contemporaneamente.

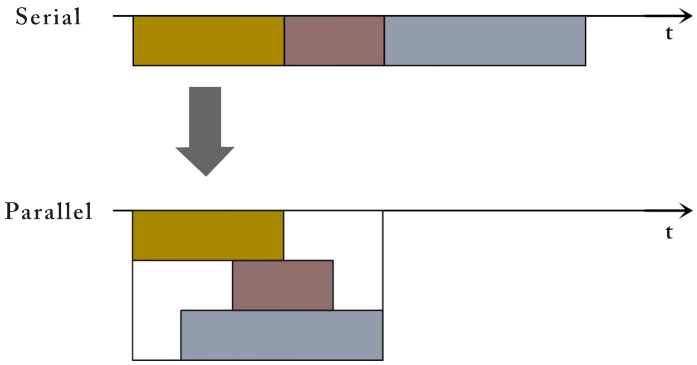

Figura 3.2: Esempio programma con tasks, ogni rettangolo rappresenta un processo indipendente.

La direttiva che permette di creare questi processi è `#pragma omp task`. Questa crea una copia indipendente di tutte le variabili utilizzate dal segmento di codice che si vuole rendere parallelo e mediante la clausola `shared(var)` vengono gestite più specificatamente come devono essere trattate alcune variabili. Questo però non basta, in quanto abbiamo la necessità di dover stoppare l'esecuzione e farlo aspettare che i vari processi aperti finiscano, visto che vengono gestiti dal sistema e non più dal programma. La libreria offre varie possibilità per fare ciò, quella che si adattava meglio al nostro caso è `#pragma omp taskwait`.

Di seguito si riporta un segmento di codice dove è stata utilizzata con successo questa tecnica al fine di velocizzarlo. Si può subito notare che in ogni task ci sono due cicli `for`, infatti anche se in un primo momento si è tentato di creare quattro tasks (uno per ciclo), si è visto che questo non era possibile. I `for` sono stati così accorpati in due tasks, uno per variabile

utilizzata nelle somme dei cicli; questo è dovuto al fatto che, anche se ogni task avrebbe il controllo diretto delle variabili usate, una volta passato il blocco ed effettuata la somma, ricomponendo così i vari pezzi, i risultati differivano da quelli esatti.

```
...
double scr=0.0;
double sci=0.0;
  for (size_t np=0; np<list.size(); np++)
  {
    ...
    #pragma omp task shared(scr)
    {
      ...
        for (size_t k=0; k<intsz; k++) scr+=...;
      ...
        for (size_t k=0; k<intsz; k++) scr+=...;
    }
    #pragma omp task shared(sci)
    {
      ...
        for (size_t k=0; k<intsz; k++) sci+=...;
      ...
        for (size_t k=0; k<intsz; k++) sci-=...;
    }
    #pragma omp taskwait
  }
  return complex<double> (scr,sci);
...
```

Codice 3.2: Esempio tasks utilizzato nel programma, i cicli sono stati accorpati a due a due per non creare conflitti di memoria.

Capitolo 4

Simulazioni e test

Il codice così modificato è stato eseguito al fine di testarne l'efficacia, sono state fatte due applicazioni, con due matrici quadrate di dimensione rispettivamente 1024 e 2000. Fin dai primi test si è riscontrata una effettiva accelerazione del codice, infatti i tempi complessivi di esecuzione del programma erano notevolmente minori.

	1024 x 1024	**2000 x 2000**
Sequenziale	2m40s	9m46s
Parallelo	1m38s	5m39s

Tabella 4.1: Tempi di esecuzione del programma.

Si può subito vedere che si ottiene per il primo caso una diminuzione del tempo impiegato del 39% circa, mentre nel secondo la percentuale cresce ulteriormente ad un 43%. Si può ipotizzare dunque che la percentuale possa ancora crescere con l'aumentare delle dimensioni della matrice adoperata.

4.1 Dati inconsistenti

Nonostante i risultati dati dai tempi di esecuzione totale fossero migliori delle aspettative, si è voluto testare il programma ulteriormente al fine di ottenere dati più approfonditi riguardo l'ottimizzazione.

In primo luogo si è cercato di confrontare i tempi di esecuzione su una singola iterazione e in secondo l'aumentare dell'efficacia a seconda dei threads aperti. In entrambi i casi non si sono potuti ottenere risultati soddisfacenti.

4.1.1 Problematica dei tempi

Si è cercato di confrontare i tempi di esecuzione di un singolo algoritmo al fine di analizzarli meglio e studiarne le differenze in base alla grandezza delle matrici. Per fare ciò è stata usata da prima la libreria standard per la misurazione dei tempi in C (`time.h`) ma si è rivelata inefficace, in quanto i tempi risultanti non combaciavano con l'effettiva esecuzione del programma. Successivamente si è provata la funzione `omp_get_wtime()` di OpenMp, la quale avrebbe dovuto misurare l'effettivo tempo di esecuzione di un'applicazione dell'algoritmo, ma pure questo metodo è stato fallimentare. Non si sono quindi ottenuti risultati concordanti e significativi dei vari tempi.

	4194304 states	**16000000 states**
Sequenziale	0.00s	0.00s
Parallelo	9.2e-04s	9.7e-04s

Tabella 4.2: Tempi di esecuzione dell'ultima iterazione del programma.

Nel caso sequenziale i tempi riportati sono stati presi utilizzando la libreria del C mentre in quello parallelo è stato utilizzato OpenMP. La precisione è visibilmente differente, quindi a prima vista potrebbero sembrare risultati attendibili. Nonostante ciò si è verificato che, mentre il tempo dato dalle librerie risulta pressoché nullo, il tempo effettivamente trascorso per l'esecuzione poteva durare anche qualche minuto.

4.1.2 Problematica threads

In OpenMP c'è la possibilità di controllore il numero di quanti threads utilizzare ad ogni parallelizzazione mediante la funzione `omp_set_num_threads(int)`. Abbiamo così cercato di fissare all'inizio del programma il numero di threads da utilizzare, per poi confrontare i tempi in modo da capire con quanti veniva massimizzata la prestazione.

Teoricamente ci si aspettava che il tempo di esecuzione diminuisse con il crescere del numero dei threas, fino poi a stabilizzarsi attorno ad un determinato valore. Abbiamo iniziato i test partendo da un singolo thread

per poi aumentare fino ad arrivare ad 8.

	2000 x 2000
Sequenziale	9m46s
#Threads 1	5m39s
#Threads 2	5m38s
#Threads 4	5m37s
#Threads 8	5m41s

Tabella 4.3: Tempi di esecuzione del programma fissando i threads.

I dati riportati sono già abbastanza significativi per descrivere il problema. Le aspettative iniziali erano che l'esecuzione del programma con un singolo thread sarebbe stata la più lenta in assoluto, visto che il tutto sarebbe stato svolto in maniera sequenziale e che per gestire i threads (anche se è solo uno) il sistema impiega del tempo. Invece da come si vede dai dati raccolti è stata notevolmente più veloce di quella sequenziale e leggermente più veloce di quella con 8 threads.

Questo ci ha portato a pensare che i tasks non siano gestibili manualmente e che non si possa fissare il massimo numero, ma che semplicemente vengano aperti all'occorrenza.

I tempi più alti ottenuti aumentando il numero dei threads si possono spiegare tenendo conto che ogni volta che viene aperto un nuovo thread il sistema impiega del tempo per allocare in memoria i dati che utilizzerà, si ha

così una sorta di tempo di preparazione; tuttavia le differenze temporali fra i vari casi sono irrisorie, il che ci porterebbe a pensare che siano solo discrepanze dovute alla gestione dei processi e del carico di lavoro da parte del sistema.

Oltre al tempo totale di esecuzione si è tentato di rilevare i tempi di un singolo algoritmo in base al numero dei threads ma per ciò che è stato detto nel paragrafo precedente i risultati furono del tutto inconcludenti.

	4194304 states	**16000000 states**
#Threads 1	9.3e-04s	9.7e-04s
#Threads 2		1.5e-02s
#Threads 4		2.8e-02s
#Threads 8	6.6e-02s	6.5e-02s

Tabella 4.4: Tempi di esecuzione dell'ultima iterazione fissando i threads.

Infatti per svolgere questo determinato calcolo si vedeva effettivamente che il computer impiegava tempi dell'ordine di minuti.

4.2 Possibili miglioramenti futuri

Il lavoro svolto, come si è visto, ha portato un ad una accelerazione sostanziale dei tempi di esecuzione totali del programma. Nonostante ciò la parallelizzazione potrebbe essere migliorata ulteriormente, infatti il programma attualmente gestisce solo massimo quattro processi simultaneamente.

In primo luogo si potrebbe pensare di nidificare altri processi all'interno di quelli già resi paralleli. Attualmente vengono eseguite in parallelo le moltiplicazioni che sono descritte all'interno delle varie funzioni `multiply(...)`. Si potrebbe pensare dunque di eseguire più funzioni `multiply(...)` contemporaneamente, cercando ovviamente di gestire le variabili in modo appropriato senza creare nuovi problemi di approssimazione, così' da sfruttare al meglio gli altri core presenti. Il computer usato per i test dovrebbe gestire senza problemi 8 processi simultanei. Questa strategia di parallelizzazione è stata tentata ma senza grandi risultati, si sono riscontrati vari problemi dovuti alla gestione delle variabili e dei processi simultanei, ma il tutto fa pensare che sia possibile attuarla.

In secondo luogo abbiamo già spiegato che c'è un tempo di preparazione dei threads ogni volta che viene richiesto al sistema di aprire una sezione parallela. Questo ci fa pensare che per moltiplicazioni di matrici piccole la parallelizzazione porti ad un tempo di esecuzione maggiore del calcolo. Infatti, se il tempo adoperato dal sistema per la gestione dei threads è dello

stesso ordine di grandezza del tempo impiegato per eseguire il calcolo, potremmo avere un breve rallentamento rispetto al tempo che si avrebbe se il programma venisse fatto girare in maniera sequenziale. Lo step successivo sarebbe dunque quello di individuare le dimensioni delle matrici per le quali una strategia di parallelizzazione è ottimale e far eseguire il programma in maniera sequenziale per i calcoli delle matrici più piccole, gestendo così la parallelizzazione delle programma a seconda della complessità del calcolo.

Per poter però apportare questi ulteriori miglioramenti sarebbe utile avere dei dati sui tempi coerenti. Quindi il primo problema da risolvere sarebbe quello del calcolo dei tempi di esecuzioni dei vari algoritmi. Visto che per ora, come è già stato spiegato prima, quest'analisi si è rivelata inconcludente.

Queste due modifiche non potranno portare ad un miglioramento sostanziale, in quanto già ora siamo arrivati ad una accelerazione di circa il 40% sui tempi totali di esecuzione. Sarebbero solo dei ritocchi finali di ottimizzazione del codice che porterebbero solo ad una diminuzione marginale dei tempi.

Capitolo 5

Conclusioni

Dai risultati dei test effettuati si è potuto riscontrare un effettivo miglioramento delle prestazioni, che si aggira attorno al 40% circa rispetto ai tempi di esecuzione della stessa simulazione con una versione del programma scritta in maniera sequenziale. Attenendoci così agli obbiettivi generali che ci eravamo prefissati.

Non è stato possibile invece, per problemi relativi alle funzioni della libreria standard del C `time.h` e della stessa OpenMP, verificare i tempi di esecuzione di una singola iterazione, quindi per calcolare l'efficacia dell'ottimizzazione eseguita ci siamo dovuti basare sul tempo totale di esecuzione della simulazione.

Un altro problema che si è riscontrato è stato quello della gestione dei threads statica, infatti non è stato possibile studiare l'algoritmo fissando il numero dei threads che verranno utilizzati a priori; i risultati di questi test infatti non mostrano nessuna modifica sostanziale sui tempi di esecuzione. Questo problema però è da attribuire ad OpenMP in quanto si è scoperto che

non è ancora in grado ottenere un controllo totale sui threads.

Infine si ritiene che altri miglioramenti di carattere secondario atti alla velocizzazione dell'algoritmo siano possibili. Per esempio si potrebbe gestire la parallelizzazione in maniera intelligente utilizzandola soli nei casi necessari, oppure l'annidamento di zone di parallelizzazione al fine di cercare di aumentare ulteriormente il numero di matrici moltiplicate contemporaneamente.

Appendice

Si riportano di seguito i segmenti del codice dove sono stati implementati gli algoritmi per la parallelizzazione.

superaction.cc

```
//____________________________________________________________________________
size_t Lanczos::thick (Superaction & superaction, Action & guess)
{
  //
  //    Lanczos diagonalization using thick restart algorithm
  //
  size_t i, j, k, m, mold, lfound, hfound;
  //
  long tm = timecpu ();
  long ta = 0;
  double t_function = omp_get_wtime ();
  //
  // machine precsion times small factor
  //
  double eps = machine_precision () * 10.0;
  //
  //    Allocate memory for Lanczos coefficients, and Krylov eigenvectors
  //
  size_t  memsize = l_steps * l_steps + 6 * l_steps;
  Storage memory (memsize * sizeof (double));
  //
  //    The matrix of eigenvectors (in the Lancos vectors base)
```

```
//
double  * y        = (double *) memory .storage ();
//
//    Lanczos coefficients (then eigenvalues)
//
double  * a        = y + l_steps * l_steps; // Ritz values
//
//    Lanczos off diagonal coefficients (residual norms)
//
double  * b        = a + 2 * l_steps;   // Lanczos offdiagonal coefficients
//
//    Service space
//
double  * c         = b + 2 * l_steps;
//
//    Final eigenvalues, tolerances, and residuals (errors of eigenvalues)
//
double  * aold     = l_value;             // eigenvalues
double  * delta    = l_tolerance;
double  * residual = l_error;
//
size_t applyes    = 0;
size_t iterations = 0;
//
size_t scalars    = 0;
size_t locals      = 0;
size_t globals    = 0;
size_t randoms    = 0;
size_t start       = 0;
size_t deflated   = 0;
size_t deflate    = 0;
size_t low        = l_found;
size_t lowextra   = 6;
size_t high        = l_found;
size_t highextra  = 6;
if (l_strategy == 1) {
  high = 0;
  highextra = l_found + 6;
}
if (l_strategy == 0)   high = highextra = 0;
```

```
size_t lwanted = low;
size_t hwanted = high;
//
//    Setup for scalar products and complex multiple
//
vector<Abinary> scalar;
binary (scalar, guess, guess);
complex<double> z;
if (guess .iscomplex ()) z = complex<double> (1.0, 1.0);
else                     z = 1.0;
Action dummy (guess .range (), z);
dummy .scalar (z);
vector<Abinary> multiple;
binary (multiple, guess, dummy, guess);
//
Lanczos & vector = *this;
double  * v;
double  * q      = vector [0];
double  * w      = guess .storage ();
size_t    dim   = guess .dimension ();
//
double alpha       = 0.0;
double beta        = 0.0;
double beta2       = 0.0;
double eta         = 0.0;
//
for (i = csize; i < l_size; i++) q [i] = w [i];
//
size_t kmax = l_steps;
deflate = deflated = 0;
mold = m = lfound = hfound = 0;
//
while (iterations++ < l_repeats) {
  //
  for (k = start; k < kmax; k++) {
    m = k + 1;
    w = vector [m];
    //
    // Initialization
    //
```

```
if (k > start) {
  q = vector [k-1];
  for (i = csize; i < l_size; i++) w [i] = -beta * q [i];
  release (k-1);
}
else {
  for (i = csize; i < l_size; i++) w [i] = 0.0;
  for (j = deflated; j < start; j++) {
    v = vector [j];
    for (i = csize; i < l_size; i++) w [i] -= b [j] * v [i];
    release (j);
  }
}
//
q = vector [k];                   // q = v_k is orthogonal to w = v_{k-1}
long te = timecpu ();
biapply (w, superaction, q);        // w  --->  w + superaction * q
ta += (timecpu () - te);
applyes++;
//
// Computing diagonal matrix element
//
alpha = 0.0;
for (i = csize; i < l_size; i++) alpha += q [i] * w [i];
if (applyes == 1) initial_value = alpha;
scalars++;
//
// Ortogonalization to obtain the residual
//
for (i = csize; i < l_size; i++) w [i] -= alpha * q [i];
eta = alpha * alpha;
if (k > start) eta += beta * beta;
else for (j = 0; j < start; j++) eta += b [j] * b [j];
//
for (i = csize, beta2 = 0.0; i < l_size; i++) beta2 += w [i] * w [i];
scalars++;
//
// Are norms near zero?
//
if (beta2 < eps) beta2 = 0.0;
```

```
if (beta  < eps) beta  = 0.0;
if (eta   < eps) eta   = 0.0;
//
// Re-orthogonalization
//
if (beta2 > eta) {
  //
  //      Local re-orthogonalization (little correction)
  //
  locals++;
  eta = 0.0;
  for (i = csize; i < l_size; i++) eta += q [i] * w [i];
  alpha -=eta;
  for (i = csize; i < l_size; i++) w [i] -= eta * q [i];
  if (k > start) {
    v = vector [k-1];
   #pragma omp parallel
     #pragma omp single
        z = multiply (v, w, scalar);
    scalars++;
    multiply (w, -z, v, multiple);
    release (k-1);
  }
  //
  //      Normalize
  //
  for (i = csize, beta2 = 0.0; i < l_size; i++) beta2 += w [i] * w [i];
  scalars++;
  if (beta2 > eps) {
    beta = sqrt (beta2);
    for (i = csize; i < l_size; i++) w [i] /= beta;
  }
  else beta = beta2 = 0.0;
}
else if (beta2 > eps * eta) {
  //
  //      Global re-othogonalization (global orthogonality correction)
  //
  globals++;
  eta = 0.0;
```

```
  for (i = csize; i < l_size; i++) eta += q [i] * w [i];
  alpha -= eta;
  for (i = csize; i < l_size; i++) w [i] -= eta * q [i];
    for (j = 0; j < k; j++) {
      v = vector [j];
     #pragma omp parallel
       #pragma omp single
          z = multiply (v, w, scalar);
      scalars++;
      multiply (w, -z, v, multiple);
      release (j);
    }
  //
  //      Normalize
  //
  for (i = csize, beta2 = 0.0; i < l_size; i++) beta2 += w [i] * w [i];
  scalars++;
  if (beta2 > eps) {
    beta = sqrt (beta2);
    for (i = csize; i < l_size; i++) w [i] /= beta;
  }
  else beta = beta2 = 0.0;
}
else beta = beta2 = 0.0;
if (beta2 <= eps) {
  //
  //      Krylov space exausted.
  //      Generate a random vector (orthogonal to actual Krylov
  //      space) to continue algorithm.
  //
  randoms++;
  random (m);
    for (j = 0; j < m; j++) {
      v = vector [j];
     #pragma omp parallel
       #pragma omp single
          z = multiply (v, w, scalar);
      scalars++;
      multiply (w, -z, v, multiple);
      if (j != k) release (j);
```

```
      }
    //
    //      Normalize
    //
    for (i = csize, beta2 = 0.0; i < l_size; i++) beta2 += w [i] * w [i];
    scalars++;
    if (beta2 > eps) {
      beta = sqrt (beta2);
      for (i = csize; i < l_size; i++) w [i] /= beta;
    }
    else beta2 = 0.0;
    beta = 0.0;
  }
  //
  // Lanczos coefficients
  //
  a [k] = alpha;
  b [k] = beta;
  if (applyes == 1) initial_residual = beta;
  //
  // A little cleaning
  //
  if ((k > start) && (b [k-1] < eps * (fabs (a [k-1]) + fabs (a [k]))))
    b [k-1] = 0.0;
  release (m);
  //
  // If no residual vector terminate
  //
  if (beta2 == 0.0) break;
  //
} // for (k = start; ...
//
//  Mark last deflated vectors with a special value for the residual
//  to recognize them in output.
//
for (j = 0; j < deflated; j++) b [j] = 7.77e-77;
//
//   Record number of deflated vectors in last iteration
//
deflate = deflated;
```

```
//
//  Diagonalization in the Krylov subspace
//
krylov (a + deflated, b + deflated, y, start - deflated, m - deflated);
//
//  Evaluate tolerances and residuals
//
for (j = deflated; j < m; j++) {
  b [j] = beta * y [m - deflated - 1 + (j - deflated) * (m - deflated)];
  delta [j] = 0.0;
  if (mold) {
    //
    //  Search for nearest old eigenvalue
    //
    delta [j] = a [m-1] - aold [deflated];
    for (i = deflated; i < mold; i++)
      if (fabs (a [j] - aold [i]) < fabs (delta [j]))
        delta [j] = a [j] - aold [i];
  }
}
//
//  Choose vectors to keep (for restart or deflating) and
//  count accepted eigenvalues (taking into account previous
//  deflated vectors)
//
lfound = lwanted - low;
hfound = hwanted - high;
size_t jlow  = deflated + low;
size_t jhigh = m - high - 1;
for (j = deflated; j < m; j++) {
  c [j] = 0.0;
  double vale = fabs (b [j]);
  if (vale < eps) {
    //
    //      Mark converged vector for deflation.
    //
    c [j] = -1;
    if (j < jlow)  {
      //
      //    accept deflated low
```

```
    low--;
    lfound++;
  }
  if (j > jhigh) {
    //
    //    accept deflated high
    high--;
    hfound++;
  }
  continue;
}
//
//  Keep lowest eigenvalues (wanted + 6)
//
if (j < jlow + lowextra) c [j] = 1.0;
if (j < jlow)  {
  //
  //      Check if vector was acceptable
  //
  double vald = fabs (delta [j]);
  if (mold == 0) vald = 1.0;        // first iteration:      don't accept
  if (m == dim)  vald = 0.0;        // no more iterations:   accept
  //
  if (vald < l_zerotolerance && vale < l_zeronorm) lfound++;
}
//
//  Keep also highest eigenvalues.
//
if (j > jhigh - highextra) c [j] = 1.0;
//
if (j > jhigh) {
  //
  //  Check if vector was acceptable
  //
  double vald = fabs (delta [j]);
  if (mold == 0) vald = 1.0;        // first iteration:      don't accept
  if (m == dim)  vald = 0.0;        // no more iterations:   accept
  //
  if (vald < l_zerotolerance && vale < l_zeronorm) hfound++;
}
```

```
}
//
//  Set the number of inner vectors to keep
//
size_t select = high + low;
//
//  Check if we need to restart
//
if ((lfound + hfound) == (lwanted + hwanted) || beta2 == 0.0) {
  //
  //  We have found all wanted or possible eigenvalues or all wanted
  //  eigenvectors are deflated or deflating.
  //  No need to restart (and to compute extra eigenvectors), unmark
  //  extra vectors.
  //
  for (j = jlow; j < m; j++) c [j] = 0;
  //
  //   Avoid next search.
  //
  select = 0;
  lfound = lwanted;
}
//
//  select some other vector
//
while (select) {
  double emin = 1.00 + beta;
  size_t jmin = m;
  for (j = deflated; j < m; j++) {
    if (c [j]) continue;
    //
    if (fabs (b [j]) < emin) {
      emin = fabs (b [j]);
      jmin = j;
    }
  }
  if (jmin == m) select = 0;
  else {
    c [jmin] = 1.0;
    select--;
```

```
  }
}
//
//  Compute selected Ritz vectors
//
select = 0;
for (k = deflated; k < m; k++) {
  if (c [k] == 0.0) continue;
  //
  select++;
  //
  w = vector [m + select];
  for (i = csize; i < l_size; i++) w [i] = 0.0;
  //
  for (j = deflated; j < m; j++) {
    v = vector [j];
    eta = y [j - deflated + (k - deflated) * (m - deflated)];
    for (i = csize; i < l_size; i++) w [i] += eta * v [i];
    release (j);
  }
  release (m + select);
}
//
//  Put vectors and values in right places
//
select = 0;
start  = deflated;
for (k = deflated; k < m; k++) {
  if (c [k] == 0.0) continue;
  //
  select++;
  //
  //  Find insert position for vectors to keep
  //
  size_t insert = start;
  if (c [k] < 0.0) {
    //
    //  Add to deflated vectors and look for insertion position
    //
    for (insert = deflated; insert; insert--)
```

```
      if (a [insert-1] <= a [k]) break;
    deflated++;
  }
  //
  //  Put vector in place
  //
  for (j = start; j > insert; j--) storage (j) << storage (j-1);
  storage (insert) << storage (m + select);
  //
  //  Put values in place
  //
  double aa = a [k];
   double bb = b [k];
  double dd = delta [k];
  for (j = k; j > insert; j--) {
    a [j]     = a [j-1];
    b [j]     = b [j-1];
    delta [j] = delta [j-1];
  }
  a [insert]     = aa;
  b [insert]     = bb;
  delta [insert] = dd;
  //
  //Update start position
  start++;
}
//
//  move residual vector in start position
//
storage (start) << storage (m);
//
//  Remove unused vectors to avoid meaningless swap reads
//
for (k = start + 1; k < m + select + 1; k++) remove (k);
//
//  Update eigenvalues and residuals for next step and in case of exit
//  from while (tolerances were reordered above)
//
mold = m;
for (j = 0; j < m; j++) {
```

```
    aold     [j] = a [j];
    residual [j] = b [j];
  }
  //
  //  Check convergengy
  //
  if ((lfound + hfound) == (lwanted + hwanted) || beta2 == 0.0) break;
  //
  //  Verify to have room for next iteration
  //
  kmax = l_steps + deflated;
  if (kmax > 2 * l_steps) kmax = 2 * l_steps;
  if (kmax - start < 2) break;
  //
} // while (iterations++ ...
//
//    Reorder eigenvalues and wanted eigenvectors
//
//    The deflated vectors and eigenvalues are ordered, but the while
//    loop may break with some eigenvectors (from deflated to
//    deflated + low) not in position, so we must take care of them.
//    Note that the true deflated vectors number of last iteration
//    is recorded in variable 'deflate', not 'deflated'.
//
for (k = deflated; k < m; k++) {
  for (j = k; j; j--) if (aold [j-1] <= a [k]) break;
  if (j < k) {
    double aa = aold     [k];
    double dd = delta    [k];
    double bb = residual [k];
    for (i = k; i > j; i--) {
      aold     [i] = aold     [i-1];
      delta    [i] = delta    [i-1];
      residual [i] = residual [i-1];
    }
    aold     [j] = aa;
    delta    [j] = dd;
    residual [j] = bb;
    if (k < deflated + low) {
      //
```

```
        //      Move k far away
        //
        storage (m) << storage (k);
        //
        //      Make room in j
        //
        for (i = k; i > j; i--) storage (i) << storage (i-1);
        storage (i) << storage (m);
      }
    }
  }
  //
  //    Remove unwanted vectors
  //
  for (k = l_found; k < 3 * l_steps + 1; k++) remove (k);
  //
  tm = timecpu () - tm;
  t_function = omp_get_wtime () - t_function;
  stringstream outp;
  outp << "Found " << lfound << "/" << lwanted;
  if (hwanted) outp << "+" << hfound << "/" << hwanted;
  outp << " vals " << applyes << "/" << iterations << " its "
       << locals << "+" << globals << "+" << randoms << " orts ";
  if (deflate) outp << deflate << " deflated ";
  cout << setw (60) << left << outp .str ()
       << " Cpu " << timecpustr (ta) << "/" << timecpustr (tm) << endl;
  cout << "Esecution time: " << t_function << "  #Threads: " << omp_get_max_threads ()
<< endl;
  //
  //    Set actual found and last iteration steps on return.
  //
  l_found = lfound;
  l_steps = m;
  return l_found;
}
//
//============================================================================


//____________________________________________________________________________
void biapply (double * mr, Superaction & superaction, double * ms,
              bool releasing)
```

```
{
  //
  //    Fast apply a superblock operator (assuming that    all is well
  //    defined) using allocated memory pointers.
  //
  double * mi = superaction .storage ();
  for (size_t n = 0; n < superaction .size (); n++) {
    Biaction & ba = superaction [n];
    double * minner  = ms;
    if (ba .ba_rgtop .storage ()) {
      memset (mi, 0, ba .ba_size * sizeof (double));
      #pragma omp parallel
        #pragma omp single
          multiply (mi, ms, 0,
                 ba .ba_rgtop .storage (), ba .ba_rgtop .sparsed (),
                 ba .ba_rgtab, true);
      if (releasing) ba .ba_rgtop .release ();
      minner = mi;
    }
    if (ba .ba_lftop .storage ()) {
      #pragma omp parallel
        #pragma omp single
          multiply (mr, ba .ba_lftop .storage (), ba .ba_lftop .sparsed (),
                 minner, 0, ba .ba_lftab);
      if (releasing) ba .ba_lftop .release ();
    }
    else
      for (size_t m = 0; m < ba .ba_size; m++) mr [m] += minner [m];
  }
}
//
//_____________________________________________________________________________
```

action.cc

```
//_____________________________________________________________________________
complex<double> multiply (double * a, double * b, vector<Abinary> & list)
{
  //
  //    Computes the scalar product of two Action's memory area
  //    (Schmidt scalar product)
  //
  double scr = 0.0;
  double sci = 0.0;
  for (size_t np = 0; np < list .size (); np++)
  {
    Abinary & p = list [np];
    size_t lftro = p .ab_lftro;
    size_t lftio = p. ab_lftio;
    size_t rgtro = p .ab_rgtro;
    size_t rgtio = p .ab_rgtio;
    size_t intsz = p .ab_intsz;
    #pragma omp task shared(scr)
    {
      if (lftro && rgtro)
        for (size_t k = 0; k < intsz; k++) scr += a [lftro + k] * b [rgtro + k];
      if (lftio && rgtio)
        for (size_t k = 0; k < intsz; k++) scr += a [lftio + k] * b [rgtio + k];
    }
    #pragma omp task shared(sci)
    {
      if (lftro && rgtio)
        for (size_t k = 0; k < intsz; k++) sci += a [lftro + k] * b [rgtio + k];
      if (lftio && rgtro)
         for (size_t k = 0; k < intsz; k++) sci -= a [lftio + k] * b [rgtro + k];
    }
    #pragma omp taskwait
  }
  return complex<double> (scr,sci);
}
//
//_____________________________________________________________________________
```

```
void multiply (double * r, complex<double> z, double * b,
               vector<Abinary> & list)
{
  //
  //    Adds to r the product z * b
  //
  double zr = z .real ();
  double zi = z .imag ();
  for (size_t np = 0; np < list .size (); np++) {
    Abinary & p = list [np];
    size_t resro = p .ab_resro;
    size_t resio = p. ab_resio;
    size_t rgtro = p .ab_rgtro;
    size_t rgtio = p .ab_rgtio;
    size_t intsz = p .ab_intsz;
    if (zr && rgtro)
      #pragma omp parallel for
          for (size_t k = 0; k < intsz; k++)
            r [resro + k] += zr * b [rgtro + k];
    if (zr && rgtio)
      #pragma omp parallel for
          for (size_t k = 0; k < intsz; k++)
            r [resio + k] += zr * b [rgtio + k];
    if (zi && rgtro)
      #pragma omp parallel for
          for (size_t k = 0; k < intsz; k++)
            r [resio + k] += zi * b [rgtro + k];
    if (zi && rgtio)
      #pragma omp parallel for
          for (size_t k = 0; k < intsz; k++)
            r [resro + k] -= zi * b [rgtio + k];
  }
}
//
//_____________________________________________________________________________
void multiply (double * p, double * a, double * b,
               vector<Abinary> & list, bool fermi)
{
  //
  //    Performs a set of multiplication of block matrices according to
```

```
  //    list, taking into account Fermi-Bose statistic if fermi is true
  //
  //    The non-scalar Actions are assumed normalized to unitary scalar
  //    factors
  //
  double zar = ((complex<double> *) a) [0] .real ();
  double zai = ((complex<double> *) a) [0] .imag ();
  double zbr = ((complex<double> *) b) [0] .real ();
  double zbi = ((complex<double> *) b) [0] .imag ();
  for (size_t np = 0; np < list .size (); np++) {
    Abinary & pr     = list [np];
    size_t  lftro  = pr .ab_lftro;
    size_t  lftio  = pr .ab_lftio;
    size_t  rgtro  = pr .ab_rgtro;
    size_t  rgtio  = pr .ab_rgtio;
    size_t  resro  = pr .ab_resro;
    size_t  resio  = pr .ab_resio;
    long    lftsz  = pr .ab_lftsz;
    long    rgtsz  = pr .ab_rgtsz;
    long    intsz  = pr .ab_intsz;
    double  factor = 1.0;
    if (fermi && (lftsz < 0) && (rgtsz * intsz < 0)) factor = -1.0;
    if (lftsz < 0) lftsz = - lftsz;
    if (rgtsz < 0) rgtsz = - rgtsz;
    if (intsz < 0) intsz = - intsz;
    if ((lftro || lftio) && (rgtro || rgtio)) {
      //
      // Non scalar * non scalar (non scalar result)
      //
      #pragma omp task shared(p)
      {
        if (lftro && rgtro && resro)
          for (long k = 0; k < intsz; k++)
            for (long j = 0; j < rgtsz; j++)
              for (long i = 0; i < lftsz; i++)
                p [resro + i + j*lftsz] += a [lftro + i + k*lftsz] * b [rgtro + k +
j*intsz] * factor;
        if (lftio && rgtio && resro)
          for (long k = 0; k < intsz; k++)
            for (long j = 0; j < rgtsz; j++)
              for (long i = 0; i < lftsz; i++)
```

```
              p [resro + i + j*lftsz] -= a [lftio + i + k*lftsz] * b [rgtio + k +
j*intsz] * factor;
      }
      #pragma omp task shared(p)
      {
        if (lftro && rgtio && resio)
           for (long k = 0; k < intsz; k++)
             for (long j = 0; j < rgtsz; j++)
               for (long i = 0; i < lftsz; i++)
                 p [resio + i + j*lftsz] += a [lftro + i + k*lftsz] * b [rgtio + k +
j*intsz] * factor;
        if (lftio && rgtro && resio)
           for (long k = 0; k < intsz; k++)
             for (long j = 0; j < rgtsz; j++)
               for (long i = 0; i < lftsz; i++)
                 p [resio + i + j*lftsz] += a [lftio + i + k*lftsz] * b [rgtro + k +
j*intsz] * factor;
      }
      #pragma omp taskwait
    }
//---------------------------------------------------------------------
    else if (rgtro || rgtio) {
      //
      // Scalar * non scalar      (non scalar result)
      //
      if (resro && rgtro)
        #pragma omp parallel for
          for (long k = 0; k < intsz; k++)
            p [resro + k] += zar * b [rgtro + k];
      if (resro && rgtio)
        #pragma omp parallel for
          for (long k = 0; k < intsz; k++)
            p [resro + k] -= zai * b [rgtio + k];
      if (resio && rgtro)
        #pragma omp parallel for
          for (long k = 0; k < intsz; k++)
            p [resio + k] += zai * b [rgtro + k];
      if (resio && rgtio)
        #pragma omp parallel for
          for (long k = 0; k < intsz; k++)
            p [resio + k] += zar * b [rgtio + k];
```

```
    }
    else if (lftro || lftio) {
      //
      // Non scalar * scalar      (non scalar result)
      //
      if (resro && lftro)
        #pragma omp parallel for
          for (long k = 0; k < intsz; k++)
            p [resro + k] += a [lftro + k] * zbr;
      if (resro && lftio)
        #pragma omp parallel for
          for (long k = 0; k < intsz; k++)
            p [resro + k] -= a [lftio + k] * zbi;
      if (resio && lftro)
        #pragma omp parallel for
          for (long k = 0; k < intsz; k++)
            p [resio + k] += a [lftro + k] * zbi;
      if (resio && lftio)
        #pragma omp parallel for
          for (long k = 0; k < intsz; k++)
            p [resio + k] += a [lftio + k] * zbr;
    }
    else
      //
      // Scalar * scalar  (scalar result)
      //
      ((complex<double> *) p) [0] +=
        complex<double> (zar * zbr - zai * zbi, zar * zbi + zai * zbr);
  }
}
//
//_____________________________________________________________________________
void multiply (double * p, double * a, size_t * ia, double * b, size_t * ib,
               vector<Abinary> & list, bool fermi)
{
  //
  //    Performs a set of multiplication of block matrices according to
  //    list, taking into account Fermi-Bose statistic if fermi is true
  //
  //    The non-scalar Actions are assumed normalized to unitary scalar
```

```
//    factors
//
double zar = ((complex<double> *) a) [0] .real ();
double zai = ((complex<double> *) a) [0] .imag ();
double zbr = ((complex<double> *) b) [0] .real ();
double zbi = ((complex<double> *) b) [0] .imag ();
for (size_t np = 0; np < list .size (); np++) {
  Abinary & pr     = list [np];
  size_t  lftro  = pr .ab_lftro;
  size_t  lftio  = pr .ab_lftio;
  size_t  lftsp  = pr .ab_lftsp;
  size_t  rgtro  = pr .ab_rgtro;
  size_t  rgtio  = pr .ab_rgtio;
  size_t  rgtsp  = pr .ab_rgtsp;
  size_t  resro  = pr .ab_resro;
  size_t  resio  = pr .ab_resio;
  long    lftsz  = pr .ab_lftsz;
  long    rgtsz  = pr .ab_rgtsz;
  long    intsz  = pr .ab_intsz;
  double  factor = 1.0;
  if (fermi && (lftsz < 0) && (rgtsz * intsz < 0)) factor = -1.0;
  if (lftsz < 0) lftsz = - lftsz;
  if (rgtsz < 0) rgtsz = - rgtsz;
  if (intsz < 0) intsz = - intsz;
  //
  //  left and right sparsed is not implemented, ignore right sparsed
  //
  if (lftsp && rgtsp) rgtsp = 0;
  //
  if ((lftro || lftio) && (rgtro || rgtio)) {
    //
    //Non scalar * non scalar (non scalar result)
    //
    #pragma omp task shared(p)
    {
      if (lftro && rgtro && resro && lftsp)
      //
      //      left sparsed (column-indexed)
      //
        for (long k = 0; k < intsz; k++) {
```

```
      double * bb = b + rgtro + k;
      for (size_t ii = ia [lftsp + k]; ii < ia [lftsp + k + 1]; ii++) {
        size_t i = ia [ii];
        double   aa = a [lftro + i + k *lftsz] * factor;
        double * pp = p + resro + i;
        for (long j = 0; j < rgtsz; j++)
          pp [j * lftsz] += aa * bb [j * intsz];
      }
    }
  else if (lftro && rgtro && resro && rgtsp)
   //
   //      right sparsed (row-indexed)
   //
    for (long k = 0; k < intsz; k++) {
      double * aa = a + lftro + k * lftsz;
      for (size_t jj = ib [rgtsp + k]; jj < ib [rgtsp + k + 1]; jj++) {
        size_t j = ib [jj];
        double   bb = b [rgtro + k + j *intsz] * factor;
        double * pp = p + resro + j * lftsz;
        for (long i = 0; i < lftsz; i++)
          pp [i] += aa [i] * bb;
      }
    }
  else if (lftro && rgtro && resro)
    for (long k = 0; k < intsz; k++) {
      double * aa = a + lftro + k * lftsz;
      for (long j = 0; j < rgtsz; j++) {
        double   bb = b [rgtro + k + j * intsz] * factor;
        double * pp = p + resro + j * lftsz;
        for (long i = 0; i < lftsz; i++) pp [i] += aa [i] * bb;
      }
    }
//
  if (lftio && rgtio && resro && lftsp)
   //
   //      left sparsed (column-indexed)
   //
    for (long k = 0; k < intsz; k++) {
      double * bb = b + rgtio + k;
      for (size_t ii = ia [lftsp + k]; ii < ia [lftsp + k + 1]; ii++) {
```

```
        size_t i = ia [ii];
        double  aa = a [lftio + i + k *lftsz] * factor;
        double * pp = p + resro + i;
        for (long j = 0; j < rgtsz; j++)
          pp [j * lftsz] -= aa * bb [j * intsz];
      }
    }
  else if (lftio && rgtio && resro && rgtsp)
  //
  //      right sparsed (row-indexed)
  //
    for (long k = 0; k < intsz; k++) {
      double * aa = a + lftio + k * lftsz;
      for (size_t jj = ib [rgtsp + k]; jj < ib [rgtsp + k + 1]; jj++) {
        size_t   j = ib [jj];
        double   bb = b [rgtio + k + j *intsz] * factor;
        double * pp = p + resro + j * lftsz;
        for (long i = 0; i < lftsz; i++) pp [i] -= aa [i] * bb;
      }
    }
  else if (lftio && rgtio && resro)
    for (long k = 0; k < intsz; k++) {
      double * aa = a + lftio + k * lftsz;
      for (long j = 0; j < rgtsz; j++) {
        double   bb = b [rgtio + k + j * intsz] * factor;
        double * pp = p + resro + j * lftsz;
        for (long i = 0; i < lftsz; i++) pp [i] -= aa [i] * bb;
      }
    }
}
//
#pragma omp task shared(p)
{
  if (lftro && rgtio && resio && lftsp)
  //
  //      left sparsed (column-indexed)
  //
    for (long k = 0; k < intsz; k++) {
      double * bb = b + rgtio + k;
      for (size_t ii = ia [lftsp + k]; ii < ia [lftsp + k + 1]; ii++) {
```

```
        size_t   i  = ia [ii];
        double   aa = a [lftro + i + k * lftsz] * factor;
        double * pp = p + resio + i;
        for (long j = 0; j < rgtsz; j++)
          pp [j * lftsz] += aa * bb [j * intsz];
      }
    }
  else if (lftro && rgtio && resio && rgtsp)
   //
   //      right sparsed (row-indexed)
   //
    for (long k = 0; k < intsz; k++) {
      double * aa = a + lftro + k * lftsz;
      for (size_t jj = ib [rgtsp + k]; jj < ib [rgtsp + k + 1]; jj++) {
        size_t   j  = ib [jj];
        double   bb = b [rgtio + k + j * intsz] * factor;
        double * pp = p + resio + j * lftsz;
        for (long i = 0; i < lftsz; i++) pp [i] += aa [i] * bb;
      }
    }
  else if (lftro && rgtio && resio)
    for (long k = 0; k < intsz; k++) {
      double * aa = a + lftro + k * lftsz;
      for (long j = 0; j < rgtsz; j++) {
        double   bb = b [rgtio + k + j * intsz] * factor;
        double * pp = p + resio + j * lftsz;
        for (long i = 0; i < lftsz; i++) pp [i] += aa [i] * bb;
      }
    }
//
  if (lftio && rgtro && resio && lftsp)
   //
   //      left sparsed (column-indexed)
   //
    for (long k = 0; k < intsz; k++) {
      double * bb = b + rgtro + k;
      for (size_t ii = ia [lftsp + k]; ii < ia [lftsp + k + 1]; ii++) {
        size_t   i = ia [ii];
        double   aa = a [lftio + i + k *lftsz] * factor;
        double * pp = p + resio + i;
```

```
          for (long j = 0; j < rgtsz; j++)
            pp [j * lftsz] += aa * bb [j * intsz];
        }
      }
    else if (lftio && rgtro && resio && rgtsp)
    //
    //      right sparsed (row-indexed)
    //
      for (long k = 0; k < intsz; k++) {
        double * aa = a + lftio + k * lftsz;
        for (size_t jj = ib [rgtsp + k]; jj < ib [rgtsp + k + 1]; jj++) {
          size_t   j  = ib [jj];
          double   bb = b [rgtro + k + j *intsz] * factor;
          double * pp = p + resio + j * lftsz;
          for (long i = 0; i < lftsz; i++) pp [i] += aa [i] * bb;
        }
      }
    else if (lftio && rgtro && resio)
      for (long k = 0; k < intsz; k++) {
        double * aa = a + lftio + k * lftsz;
        for (long j = 0; j < rgtsz; j++) {
          double   bb = b [rgtro + k + j * intsz] * factor;
          double * pp = p + resio + j * lftsz;
          for (long i = 0; i < lftsz; i++) pp [i] += aa [i] * bb;
        }
      }
  }
  #pragma omp taskwait
}
else if (rgtro || rgtio) {
  //
  //Scalar * non scalar      (non scalar result)
  //
  if (resro && rgtro)
    #pragma omp parallel for
      for (long k = 0; k < intsz; k++) p [resro + k] += zar * b [rgtro + k];
  if (resro && rgtio)
    #pragma omp parallel for
      for (long k = 0; k < intsz; k++) p [resro + k] -= zai * b [rgtio + k];
  if (resio && rgtro)
```

```
        #pragma omp parallel for
          for (long k = 0; k < intsz; k++) p [resio + k] += zai * b [rgtro + k];
      if (resio && rgtio)
        #pragma omp parallel for
          for (long k = 0; k < intsz; k++) p [resio + k] += zar * b [rgtio + k];
    }
    else if (lftro || lftio) {
      //
      // Non scalar * scalar      (non scalar result)
      //
      if (resro && lftro)
        #pragma omp parallel for
          for (long k = 0; k < intsz; k++) p [resro + k] += a [lftro + k] * zbr;
      if (resro && lftio)
        #pragma omp parallel for
          for (long k = 0; k < intsz; k++) p [resro + k] -= a [lftio + k] * zbi;
      if (resio && lftro)
        #pragma omp parallel for
          for (long k = 0; k < intsz; k++) p [resio + k] += a [lftro + k] * zbi;
      if (resio && lftio)
        #pragma omp parallel for
          for (long k = 0; k < intsz; k++) p [resio + k] += a [lftio + k] * zbr;
    }
    else
      //
      // Scalar left * scalar right       (scalar result)
      //
      ((complex<double> *) p) [0] +=
        complex<double> (zar * zbr - zai * zbi, zar * zbi + zai * zbr);
 }
}
//
//_____________________________________________________________________________
void multiply (Action & result, const Action & a, const Action & b,
               bool fermi)
{
  //
  //    Compute result = a * b
  //
  vector<Abinary> list;
```

```
  //
  //    use an intermediate Action to take into account the cases
  //    result == a or result == b
  //
  Action r;
  if (a .isscalar ()) {
    r = b;
    r *= a .scalar ();
    result << r;
    return;
  }
  if (b .isscalar ()) {
    r = a;
    r *= b .scalar ();
    result << r;
    return;
  }
  r = Action (a, b);
  binary (list, r, a, b);
  r .storage (r .size ());
  if (r .size () > csize) r .scalar  (a .scalar () * b .scalar ());
  else                    r .scalar  (0.0);
  double * ma = a .storage ();
  double * mb = b .storage ();
  double * mr = r .storage ();
  #pragma omp parallel
    #pragma omp single
      multiply (mr, ma, mb, list, fermi);
  r .statistic (a .statistic () * b .statistic ());
  result << r;
}
//
//_____________________________________________________________________________
complex<double> multiply (const Action & bra, const Action & ket)
{
  //
  //    Compute scalar product <bra|ket> (Schmidt scalar product)
  //
  vector<Abinary> scalar;
  //
```

```
  //    Prepare Abinary list
  //
  binary (scalar, bra, ket);
  //
  //    Effective scalar product of memory areas
  //
  complex<double> product;
  #pragma omp parallel
    #pragma omp single
      product = multiply (bra .storage (), ket .storage (), scalar);
  product *= conj(bra .scalar ()) * ket .scalar ();
  return product;
}
//
//============================================================================
```

Bibliografia

Chapman B., Jost G., van der Pas R., *Using OpenMP. Portable Shared Memory Parallel Programming*, The MIT Press, 2007;

Morandi G., Ercolessi E., and Napoli F., *Statistical Mechanics: An Intermediate Course*, World Scientific 2001;

White S.R., *Density-matrix algorithms for quantum renormalization groups*, *Phys*. Rev. B 1993, 48, 10345;

<http://openmp.org/>.