UNIVERSITÀ DEGLI STUDI DI
ROMA TOR VERGATA

LAUREA MAGISTRALE IN
COMPUTER AND INFORMATION ENGINEERING

Sp3MM for AMG: Sparse triple Matrix Multiplication for AlgebraicMultiGrid

---

Studente: Andrea Di Iorio
Matricola: 0277550

Relatore: Salvatore Filippone

Dedico questo lavoro di tesi

Ai miei genitori Rosaria e Antonio
e al mio caro fratello Enrico,
che mi hanno supportato costantemente in ogni modo

Ai miei nonni,Vincenzina e Michele,
che mi sono stati sempre accanto
e a Domenico e Annina che sono saliti in cielo

Alla mia meravigliosa compagna Cynthia,
che mi è stata vicino in questo periodo difficile

Sommario

Problema affrontato
Il problema affrontato in questo lavoro di tesi è la realizzazione di implementazioni parallele ed efficienti per il triplo prodotto tra matrici sparse, integrabili nella fase di setup dei solutori iterativi di sistemi lineari sparsi basati su metodi *Algebraic Multi Grid* o *AMG*.
L'esigenza di risolvere efficientemente sistemi lineari sparsi di grandi dimensioni insorge da applicazioni di calcolo scientifico come la risoluzione numerica di PDE su domini 2D o 3D, mediante griglie di discretizzazione, come descritto in 1.2.
La sparsità dei sistemi lineari di tipo $Ax = b$ da risolvere è sfruttabile rappresentando della matrice A in un formato sparso dove vengono omessi i valori nulli, risparmiando memoria e limitando il processamento ai soli valori non zero, dando un vantaggio notevole rispetto a tecniche algebriche convenzionali per matrici dense.
La necessità di utilizzare solutori iterativi per questo tipo di sistemi in luogo di solutori diretti, è dovuta alle proprietà di sparsità dei risultati intermedi, come verrà analizzato in 1.2.2 e mostrato per alcuni casi in figura 1.2.
Al fine di ottimizzare l'esecuzione del prodotto di Galerkin (descritto dalla formula 1.2) nella fase di setup dei metodi *AMG* ho realizzato varie implementazioni per il triplo prodotto tra matrici sparse o **Sp**arse**3M**atrixMatrix**M**ultiplication (*Sp3MM*) supportando la possibilità di inserire il lavoro in progetti fortran come `AMG4PSBLAS` (1.3), su cui è stato avviato un lavoro di integrazione.

Tecniche utilizzate
Il design delle implementazioni parallele per Sp3MM è stato guidato dalla formulazione row-by-row per il prodotto tra matrici sparse o
Sparse**M**atrixMatrix**M**ultiplication (*SpMM*, descritta in 2.2) introdotta da Gustavson [Gus78], che consiste nel calcolare la i-esima riga del prodotto $A \cdot B = C$ come: $c_{i*} = \sum_{k \in I_i(A)} a_{ik} * b_{k*}$.
Al fine di realizzare efficientemente le moltiplicazioni scalari tra gli elementi di A e le righe di B: $a_{ik} * b_{k*}$ è stato utilizzato un approccio basato su un accumulatore denso, descritto in 2.5.3.1, dati gli ottimi risultati riscontrabili in una ricerca precedente [MMAPD08].
Il triplo prodotto è stato realizzato sia combinando una coppia di operazioni di SpMM, sia direttamente estendendo la formulazione row-by-row con un approccio derivato dal lavoro di [JPD15], come descritto in 4.3.3

Per la rappresentazione delle matrici sparse utilizzate è stato usato il formato **C**ompressed**S**parse**R**ow, dato che è considerabile un formato *General-purpose* per matrici sparse, con un'occupazione in memoria proporzionale al numero di elementi non zero, ed è correntemente impiegato in un'implementazione seriale di

SpMM in AMG4PSBLAS.
Tutte le implementazioni di Sp3MM sono state scritte in C e per supportarne efficacemente una realizzazione parallela ho utilizzato le implementazioni offerte da GCC delle direttive OpenMP.
OpenMP comprende la specifica di un insieme di: direttive per compilatori, routines di libreria e variabili d'ambiente, utilizzabili per programmare ad alto livello regioni parallele in codici C/C++ e Fortran.

Le realizzazioni parallele del prodotto tra matrici sparse necessitano di una fase iniziale, tipicamente denominata fase simbolica, dove viene calcolata la dimensione del risultato finale (o un suo bound) per evitare di effettuare allocazioni dinamiche durante l'esecuzione parallela (di cui sono analizzate le problematiche a riguardo in 2.3). La fase simbolica di SpMM può essere di tipo UpperBound, determinando un limite superiore al numero di elementi non zero del risultato finale, o di tipo accurato per una determinazione esatta.
Per effettuare la fase simbolica accurata del prodotto tra matrici sparse (descritta nel capitolo 3), sono state utilizzate strutture di ricerca efficienti come RedBlack (descritti in 3.2.2) o vettori di bitmaps (descritti in 3.2.3 e in 5.3).
Al fine di poter utilizzare un'implementazione efficiente dei RedBlack Tree ho eseguito un porting in userspace dei RedBlack standard e left-cached disponibili dal kernel linux 5.10.85, come descritto in 5.2.
Implementazioni di SpMM basate su una fase simbolica di tipo UpperBound necessitano di uno spazio temporaneo per i risultati intermedi (come descritto in 3.2.5), che è stato pre-allocato all'esecuzione parallela ed assegnato atomicamente ai vari thread mediante vari approcci basati su direttive openMP o la built atomica di GCC `__atomic_fetch_add` come descritto in 5.6.

Per supportare l'integrazione efficiente delle implementazioni di Sp[3]MM realizzate in un progetto Fortran come amg4psblas, ho gestito lo shifting degli indici degli elementi non zero (a base 1 per matrici provenienti da implementazioni Fortran) con la generazione a tempo di pre-processamento di due versioni di ogni funzione che usi (porzioni del) vettore JA del formato CSR.
Le diverse versioni delle funzioni in oggetto sono generate a tempo di compilazione mediante un approccio, descritto in 5.1, basato su inclusioni multiple di porzioni di un codice sorgente generico e sulla modifica di alcune macro di configurazione.
Oltre a supportare una doppia indicizzazione, questo approccio è stato esteso a supportare in modo generico la generazione di molteplici versioni di una funzione nel prodotto simbolico in 5.1.2, dove sono state derivate a tempo di preprocessamento otto versioni di una funzione base per supportare diverse tipologie di prodotto simbolico tra matrici.

Posizionamento rispetto a risultati esistenti
Nel capitolo 2 vengono esposti vari algoritmi per effettuare le operazioni di SpMM

e Sp3MM.
Questo lavoro di tesi si colloca tra le formulazioni row-by-row del problema, sia con fase simbolica accurata che con fase simbolica di tipo UpperBound.
Vari lavori di ricerca come [DA20], visualizzano l'assegnamento dei task per la risoluzione di SpMM in 3 dimensioni (come mostrato in figura 2.6) ed in questa tesi sono state realizzate implementazioni sia monodimensionali che bidimensionali.

Risultati conseguiti
Nel capitolo 6 vengono analizzate varie configurazioni a runtime (come il tipo di scheduling openMP) e compile time delle implementazioni prodotte, confrontando le prestazioni ottenute da un'implementazione seriale di riferimento con le performance misurate nelle versioni parallele dell'operazione di Sp3MM.
Gli input per testare le varie implementazioni sono matrici sparse derivate dall'applicazione di alcune configurazioni degli AMG per risolvere un sistema lineare ottenuto da una discretizzazione alle differenze finite di una PDE di secondo ordine utilizzando una griglia regolare e le condizioni al contorno di Dirichlet. Alcune visualizzazioni delle matrici di test sono disponibile nelle figure 6.1 6.2.
In base al tipo di input utilizzato è stato riscontrato una variazione nella migliore configurazione e migliore implementazione di Sp3MM, come visibile nel grafico 6.6. Tuttavia, sono state osservate alcune configurazioni ed implementazioni tendenzialmente migliori di altre come:

- l'uso di vettori di bitmap a 64 bit (6.5) in luogo di 128 bit.
- l'uso di un assegnamento dinamico dello spazio intermedio per implmentazioni monodimensionali di SpMM di tipo UpperBound(6.4)
- l'uso del triplo prodotto diretto come implementazione per Sp3MM (come osservato in 6.4)

Confrontando per ogni matrice le prestazioni della migliore configurazione della migliore implementazione realizzata con un implementazione seriale di riferimento, si è evidenziato il vantagio di utilizzare implementazioni parallele per il problema in oggetto , come visibile nel grafico 6.7.
Variando il grado di parallelismo delle implementazioni parallele su alcuni input si è potuto osservare come la migliore configurazione non è sempre quella con il numero di thread massimo, come è possibile notare dai grafici in figura 6.9, ma è dipendente dalla dimensione della matrice e dal suo pattern di sparsità dei non zeri.

Indice

CAPITOLO 1

INTRODUZIONE

1.1 High Performace Computing per Exascale

Nel contesto odierno di incredibile sviluppo tecnologico, alcuni calcolatori come il supercomputer Fugako giapponese [1], hanno raggiunto performance di picco superiori a 1 Exaflops/s [2] ovvero 10^{18} operazioni floating point per secondo.

Risultati incredibili come questo sono legati alle enormi potenzialità offerte dal Hardware moderno, che devono essere correttamente sfruttate dal Software eseguito, in particolar modo riguardo il livello di parallelismo offerto.

1.1.1 Necessità di multi/many core nei calcolatori moderni

Il progresso tecnologico odierno nella integrazione dei componenti dei processori ha raggiunto un livello tale da consentire, a livello teorico, la creazione di unità di calcolo ulteriormente più dense di transistor (e quindi inderettamente più veloci) di quelle attualmente in commercio, ma non realizzabili a causa del surriscaldamento che verrebbe generato durante il loro utilizzo.
Per avere un'idea del problema del surriscaldamento nei processori, è possibile analizzarne la densità di potenza $\left[\frac{W}{cm^2}\right]$ comparandola con le densità raggiunte da altre entità come in figura 1.1.

[1] con 7,630,848 cores A64FX 48C da 2.2GHz
[2] in precisione singola o ulteriormente ridotta con il benchmark HPL-AI

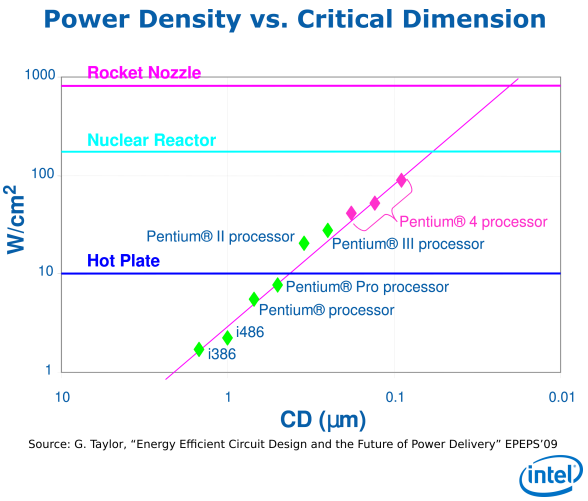


FIGURA 1.1: Comparazione tra la densità di potenza di alcuni dispositivi tecnologici con dei processori intel rispetto alla loro dimensione critica, è possibile notare come la i valori raggiunti da alcuni modelli pentium si avvicinino a quelli di un reattore nucleare

---

Una buona approssimazione della potenza dissipata da un processore è data da: $P_{cpu} = P_{dyn} + P_{sc} + P_{leak}$ dove

1. P_{dyn} modella la dissipazione di energia dovuta all'attività interna dei gate logici della CPU, approssimabile con $P_{dyn} = CV^2f$

2. P_{sc} modella la dissipazione di energia dovuta ad un collegamento diretto che si può creare tra la sorgente di energia e il ground durante un cambiamento di stato di un gate CMOS

3. P_{leak} modella le perdite di energia intrinseche dei singoli transistors.

Dato che la frequenza di lavoro dell'unità di calcolo è proporzionale ai punti 1 e 2, è necessario limitarla, purtroppo, per evitare un surriscaldamento critico dei componenti elettronici.
Indirettamente, questo comporta la necessità di limitare anche le potenzialità offerte da una singola unità di calcolo.

La soluzione per far fronte a questa limitazione fisica/tecnologica per poter costruire processori più performanti è quella di realizzare CPU con diverse unità di calcolo separate o core.
In questo contesto si inserisce il calcolo parallelo, ovvero delle tecniche per realizzare software che sfrutti la disponibilità dei vari core del calcolatore mediante un partizionamento del lavoro in parti assegnabili alle singole unità di calcolo.
L'High Performance Computing si riferisce a tecnologie hardware e software usate

per realizzare sistemi di elaborazione ad alte prestazioni per applicazioni che richiedono computazioni molto onerose, non risolvibili con sistemi informatici tradizionali.

1.2 Uso di Metodi basati su Algebraic Multi Grid nel calcolo scientifico

Applicazioni HPC tipiche sono provenienti dal mondo ingegneristico e dal calcolo scientifico, come ad esempio la risoluzione di equazioni differenziali alle derivate parziali (*PDE*) su dominii 2D o 3D.
Le PDE sono usate per descrivere moltissimi fenomeni fisici relativi alla dinamica dei fluidi, alla meccanica quantistica e chimica computazionale così come vari problemi nell'area dell'ingegneria, biologia e economia.
Dato che molto spesso non esistono soluzioni esplicite in forma chiusa a queste equazioni, vengono calcolate soluzioni approssimante ottenute da tecniche numeriche.
Molte tecniche numeriche per risolvere delle PDE trasformano le equazioni differenziali in equazioni algebrice, mediante un processo di discretizzazione delle equazioni nel dominio del problema da risolvere.
Conseguentemente un problema ricorrente in varie applicazioni algebriche e nel calcolo scientifico in generale, è risolvere sistemi sparsi di dimensioni grandi derivanti da metodi di risoluzione numerici di PDE.

Il sistema lineare sparso di riferimento che verrà usato nel seguito è

$$Ax = b \tag{1.1}$$

dove A è una matrice sparsa di grandi dimensioni, simmetrica e definita positiva *s.p.d.*

1.2.1 Importanza della sparsità delle matrici

La proprietà di sparsità della matrice A nel sistema 1.1, ovvero la proprietà di avere per la maggioranza degli elementi il valore zero, deriva dalla discretizzazione delle PDE nel problema originario in una mesh di punti [MBK05]. Una caratteristica comune nei metodi di discretizzazione utilizzabili come differenze finite, elementi finiti e volumi finiti è che il numero di elementi in ogni equazione discretizzata dipende da proprietà topologiche locali alla discretizzazione e non dalla dimensione globale del dominio del problema da risolvere [SF17; LeV07; QV94; Pat80]. Conseguentemente la matrice generata da questi metodi è comunemente sparsa.

La proprietà di sparsità di A consente l'utilizzo di rappresentazioni idonee in memoria durante il calcolo, dove vengono salvati solamente gli elementi non zero.
Formati tipici di memorizzazione per matrici sparse sono: *CompressedSparseColumn, CompressedSparseRow, Ellpack, DIA*.

Sfruttare la sparsità delle matrici nella risoluzione di un problema come 1.1, consente di ottenere svariati vantaggi rispetto ad usare le tecniche convenzionali dell'algebra lineare densa, come un utilizzo molto più efficiente della memoria.
Spingendo al limite le potenzialità dei super calcolatori attuali e cercando soluzioni sempre più accurate ai problemi moderni di calcolo scientifico, si rende indispensabile sfruttare questa proprietà quando presente, dato che tecniche classiche per matrici dense necessiterebbero di una quantità di memoria quadratica nella dimensione della matrice, contro una quantità lineare nel numero dei non zeri con tecniche di algebra sparsa.

1.2.2 Tecniche risolutive efficienti di sistemi sparsi

Solutori di sistemi come 1.1, possono essere di tipo diretto o iterativo.
I solutori di tipo diretto tipicamente impiegano una fattorizzazione della matrice A in matrici di struttura semplice (triangolare, diagonale, ortogonale), per poi applicare operazioni che portano alla soluzione del sistema x^*.
I solutori di tipo indiretto generano una sequenza di soluzioni intermedie, (ipoteticamente) convergenti alla soluzione del sistema x^*.
Una differenza significativa tra le due classi di soluzioni è che i solutori diretti non sfruttano la sparsità della matrice in input nei passaggi intermedi. Infatti, in misura differente in base al metodo di risoluzione diretto utilizzato, la fattorizzazione della matrice può portarne alla perdita della proprietà di sparsità come mostrato dalla figura 1.2.

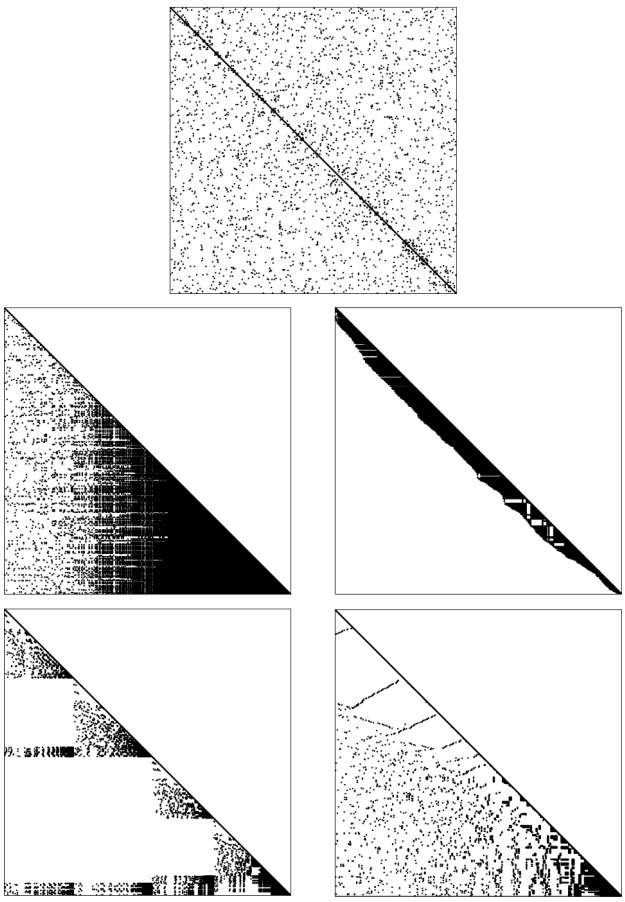


FIGURA 1.2: Varie fattorizzazioni di una matrice altamente sparsa A 500x500 con soli 3502 elementi non zero (in alto) in matrici con un numero di non zero incrementato a vari livelli [MBK05]:

- 36000 non zero con il metodo di *Cholesky* in alto a sinistra
- 14000 non zero con il metodo di *Cuthill-McKee* in alto a destra
- 6203 non zero con il metodo *minimum degree ordering* in basso a destra
- 7142 non zero con il metodo *nested dissection method* in basso a sinistra

---

Di contro a questa penalità dei solutori diretti, c'è il vantaggio che la fattorizzazione della matrice A, può essere riutilizzata per sistemi in cui viene cambiato il lato destro di 1.1, come analizzato in [MBK05].

1.2.3 Solutori MultiGrid

Tra i solutori iterativi più efficienti ci sono quelli basati su metodi *MultiGrid*, che godono di ottime caratteristiche di scalabilità come avere un numero di iterazioni e un costo per iterazione approssimatamente costante all'incrementare della dimensione del problema [XZ17; JPD15].
L'idea principale dei metodi MultiGrid è quella di risolvere il sistema 1.1 iterativamente, applicando una serie di correzioni alle soluzioni approssimate ottenute con un metodo iterativo basico.
Le correzioni sono applicate ai residui associati agli errori delle soluzioni intermedie approssimate, risolvendo un sotto problema dell'originale contente un sottoinsieme delle incognite.
Il sotto problema può essere risolto a sua volta applicando ricorsivamente delle correzioni a dei sotto problemi ulteriori, sempre più piccoli e facili da risolvere, fino a quando la risoluzione dell'ultimo sotto problema è di una difficoltà trascurabile rispetto a generarne un altro.

Le iterazioni di questi metodi possono essere strutturati con cicli di varia natura. Tra le metodologie principalmente utilizzati ci sono cicli a forma V, F e W.

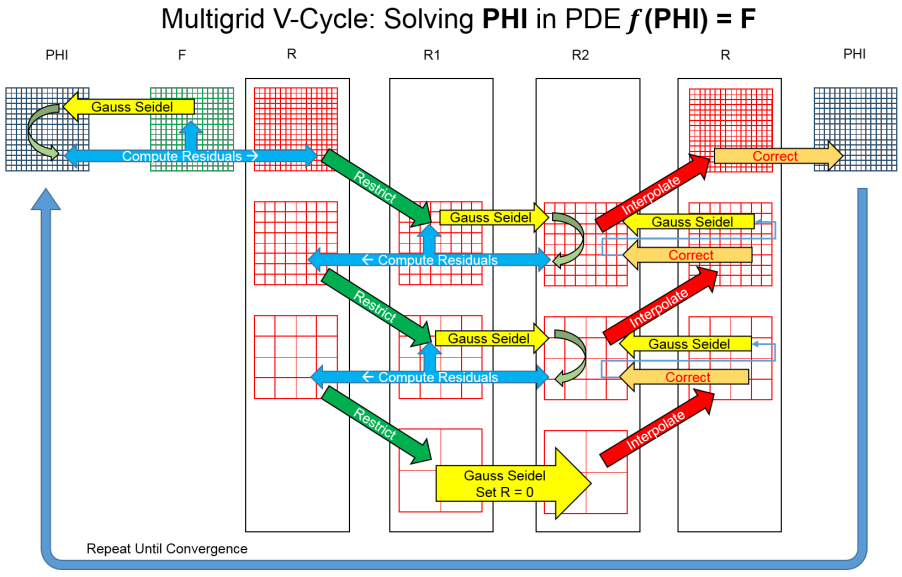


FIGURA 1.3: Rappresentazione grafica di un ciclo V di un solutore MultiGrid

Tra i principali tipi di metodi MultiGrid disponibili ci sono quelli di tipo Geometrico (*GMG*) e quelli di tipo Algebrico (*AMG*). La differenza sostanziale è il modo di

determinazione dei sotto problemi intermedi da risolvere ad ogni iterazione.
Nei metodi Geometrici, si usa una gerarchia di griglie iterativamente più sparse, derivate da informazioni sulla struttura della griglia di discretizzazione originaria.
I metodi Algebrici riescono ad utilizzare unicamente informazioni derivanti dalla matrice associata al sistema da risolvere 1.1, pur mantenendo la stessa complessità asintotica dei *GMG*.
Questa proprietà degl'*AMG*, rende possibile applicarli alla risoluzione di problemi discretizzati con griglie non strutturate, ad esempio mediante il metodo degli elementi finiti.
[XZ17].

La gerarchia di sotto-problemi intermedi da risolvere ad ogni iterazione con i metodi AMG è derivata dal triplo prodotto di Galerkin:

$$A_{l+i} = (P_l)^T \cdot A_l \cdot P_l \; l = 0, \ldots, nl - 1 \tag{1.2}$$

dove $A_0 = A$ [DDF21] e P_l è una sequenza di matrici di prolungamento $n_l \times n_{l+1}$ con $n_0 = n$ e $n_{l+1} < n_l$.

In generale il tempo complessivo di risoluzione del sistema lineare con dei metodi AMG è $T_{tot} = T_{setup} + T_{it} \cdot N_{it}$, dove $T_{it} \cdot N_{it}$ è associato al costo delle iterazioni del metodo, mentre T_{setup} comprende il calcolo di tutto il necessario all'esecuzioni delle iterazioni, tra cui la costruzione della gerarchia di sotto problemi mediante il triplo prodotto.

Realizzare il triplo prodotto tra matrici sparse, denominato **Sp**arse**3M**atrixMatrix**M**ultiplication (*Sp3MM*) in parallelo è l'obbiettivo di questa tesi, al fine di potenziare progetti come [DDF21] che sfruttano solutori basati su AMG.

1.3 Algebraic MultiGrid for Parallel Sparse BLAS

PSBLAS è una libreria software che punta a consentire la realizzazione di varie applicazioni di calcolo scientifico computazionalmnte intense, mediante implementazioni di risolutori iterativi per sistemi lineari sparsi usando il padarigma di programmazione per sistemi a memoria distribuita.
L'ambito di questa libreria è relativo alle PDE discretizzate mediante differenze finite o elementi finiti, ma anche a problemi di ottimizzazione non lineare come problemi di controllo ottimo [SF19].

AMG4PSBLAS [SF21] fornisce i metodi MultiGrid di tipo algebrico precedentemente descritti, utilizzando come layer sottostante PSBLAS.
Il progetto Software è scritto in Fortran 2003, sfruttandone le possibilità di usare un design object-oriented degli algoritmi, mediante varie funzionalità disponibili nello standard del linguaggio, consentendo un'ottima gestione delle varie implementazioni offerte.

CAPITOLO 2

ALGORITMI PER SPMM ESISTENTI

2.1 Notazione

Nel seguito si utilizzerà la seguente notazione.

Considerando il prodotto tra 2 matrici sparse, si indicherà con $C \in \mathbb{R}^{m \times n}$ il risultato del prodotto delle matrici sparse $A \in \mathbb{R}^{m \times k}$, $B \in \mathbb{R}^{k \times n}$. Dove:

$a_{ij} \in A$ è l'elemento della riga i e colonna j della matrice A.
a_{i*}, a_{*j} indicano rispettivamente la riga i-esima e la colonna j-esima della matrice A
$a_{i\ c_z - c_w} = \{a_{ij} \in a_{i*} : \ j \in [c_z, c_w]\}$ indica:
la partizione delle colonne nel range $[c_z, c_w]$ della riga i-esima della matrice A.
$\hat{C}$ indica una struttura contenete risultati intermedi al prodotto $A \cdot B$.

$I_i(A)$ indica il set di indici di colonna
relativi agli elementi non zero della riga i-esima di A
$I^j(B)$ indica il set di indici di riga
relativi agli elementi non zero della colonna j-esima di B
$I(i, j)$ indica il set di indici k di righe di A e di colonne di B tali che $a_{ik} * b_{kj} \neq 0$
$nnz(A)$ indica il numero di non zeri della matrice A.
$nzc(A)$, $nzr(A)$ indicano rispettivamente il numero di colonne e di righe di A con almeno un elemento non zero

Con Sp3MM si indicherà il triplo prodotto tra matrici sparse: $AC_{i+1} = R \cdot AC_i \cdot P$
dove:
$AC_{i+1} \in \mathbb{R}^{n_{i+1} \times n_{i+1}}$ $R \in \mathbb{R}^{ni+1 \times m}$ $AC_i \in \mathbb{R}^{n^i \times n^i}$ $P \in \mathbb{R}^{m \times n^{i+1}}$, e $i < i+1$.

Con moltiplicazione scalare si intenderà il prodotto tra uno scalare $a \in \mathbb{R}$ per un vettore $v \in \mathbb{R}^n$

2.2 Formulazioni SpMM

Gli algoritmi per la risoluzione del prodotto tra due matrici sparse possono essere classificati in base alle formulazioni e partizionamento dei dati del problema SpMM [Gao+19].
Le principali formulazioni di SpMM sono:

$$\text{Row-by-row} \qquad c_{i*} = \sum_{k \in I_i(A)} a_{ik} * b_{k*}$$

$$\text{Col-by-col} \qquad c_{*j} = \sum_{k \in I^j(B)} a_{*k} * b_{kj}$$

$$\text{Inner-product} \qquad c_{ij} = \sum_{k \in I(i,j)} a_{ik} * b_{kj}$$

$$\text{Outer-product} \qquad C = \sum_{i=1}^{k} a_{*i} \otimes b_{i*}$$

2.2.1 Rappresenzione mediante grafi

Le formulazioni per SpMM possono essere rappresentate graficamente mediante grafi a 3 strati U, W, V [BG08; Coh98] dove:

- i nodi in $U \ni u_i$: rappresentano le righe di A
- i nodi in $V \ni v_j$: rappresentano le colonne di B
- i nodi in $W \ni w_k$: rappresentano la dimensione in comune tra A e B

esiste un arco $(u_i, w_l) \; \forall \; a_{i,l} \neq 0$
esiste un arco $(w_l, v_j) \; \forall \; b_{l,j} \neq 0$
La rappresentazione mediante grafi delle formulazioni di SpMM, consente di visualizzarne le operazioni di prodotto mediante coppie di vertici (u_i, v_j) connessi da un nodo comune w_k, evidenziandone il tipo di accesso agli elementi delle matrici di input.

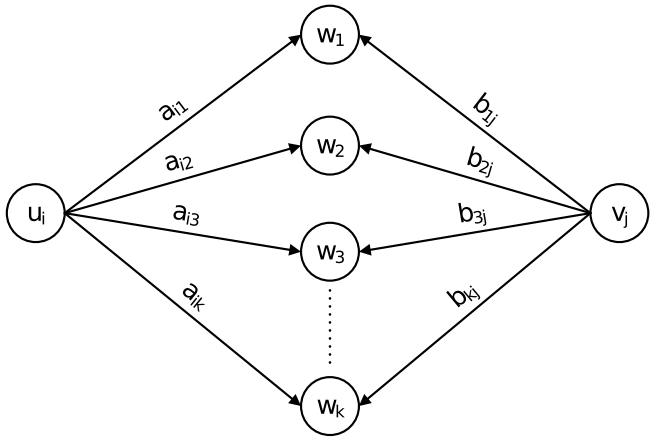


FIGURA 2.1: Rappresentazione grafica del calcolo di $c_{i,j}$ mediante Inner-product

---

Per la formulazione Inner-product è necessario esaminare ogni coppia di vertici in U e V per trovare il sottoinsieme di nodi $\tilde{W}_{ij} \subseteq W$ che connettono coppie di nodi, per poi accumulare i prodotti $a_{il} \cdot b_{lj}$ relativi ai nodi $w \in \tilde{W}_{ij}$ in c_{ij}.
In questa formulazione è possibile osservare come la sparsità delle matrici delle matrici A e B non è sfruttata dato che è necessario analizzare tutte le coppie di nodi (u_i, v_j)

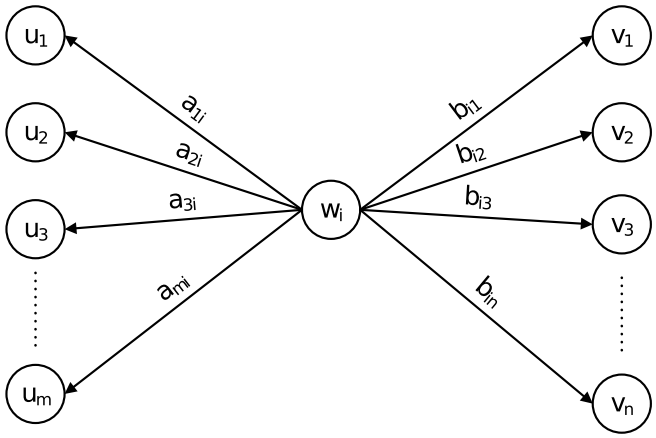


FIGURA 2.2: Rappresentazione grafica della componente $a_{*i} \cdot b_{i*}$ del calcolo di C mediante Outer-product

---

Per la formulazione Outer-product è possibile identificare coppie di vertici connesse a partire dai nodi di W che, per matrici sufficientemente sparse, può essere un'operazione

più veloce rispetto all'Inner-product.

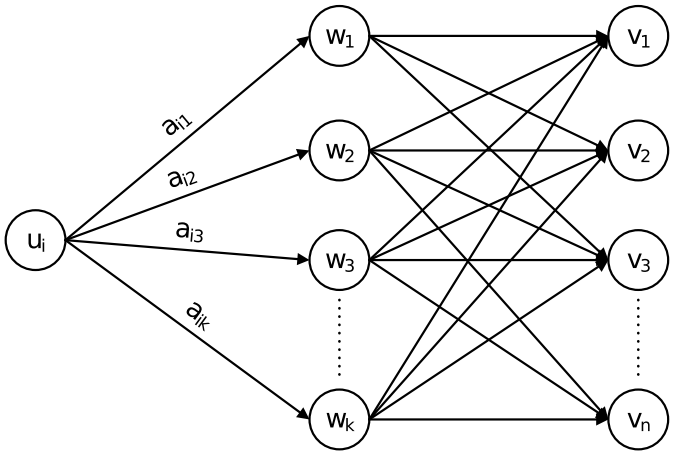


FIGURA 2.3: Rappresentazione grafica della componente $a_{i*} \cdot B$ del calcolo di C mediante formulazione row-by-row

Per la formulazione Row-by-row le coppie di vertici d'interesse per il risultato di C sono identificabili a partire da attraversamenti del grafo indipendenti a partire dai vertici di U verso V.
Nel caso col-by-col vi è un attraversamento del grafo dai nodi di V a U, isomorfico rispetto al caso Row-by-row

2.3 Determinazione della dimensione della matrice risultante

Una caratteristica comune degli algoritmi per la moltiplicazione parallela di matrici sparse è quella di preferire pre-allocazioni per C e $\hat{C}$ invece di usare (ri)allocazioni dinamiche durante la computazione parallela del prodotto.
È possibile giustificare questa tendenza considerando genericamente algoritmi paralleli implementati con direttive OpenMP come `#pragma omp parallel for`, che permettono di effettuare una parallelizzazione di cicli, noto precedentemente il numero di iterazioni da suddividere tra i thread.
In questo contesto, delle allocazioni dinamiche richiedono la possibilità di poter uscire precedentemente dal ciclo parallelizzato data la possibilità di fallimento di una (ri)allocazione, il che è contrario alla filosofia di parallelizzazione OpenMP (per quanto siano disponibili costrutti per una uscita anticipata, che però richiedono una speciale configurazione, disabilitata di default per motivi di performance [Boa20]
)

Si considereranno quindi solo implementazioni per SpMM che calcolino il numero di non zero di C e $\hat{C}$ (o un loro bound) per effettuarne una pre-allocazione.

Il numero di non zero delle righe del risultato dell'operazione di SpMM può essere determinato:
sia con un UpperBound con un basso costo computazionale
sia con in una modalità accurata con maggiore costo computazionale.

Il calcolo (di un bound superiore) del numero di numero di non zeri di C e $\hat{C}$ viene tipicamente denominato Prodotto Simbolico o fase simbolica, e il calcolo effettivo degli elementi non zero Prodotto Numerico o fase numerica.

2.3.1 UpperBound

Un UpperBound sulla lunghezza di una riga risultante a SpMM è computabile come riportato nell'algoritmo 4 di [Gao+19].
In questa soluzione, ispirandosi a una formulazione row-by-row di SpMM, si determina la dimensione massima di ogni riga della matrice risultante come:
$|I_i(C)| \leq \sum_{j \in I_i(A)} |I_j(B)|$.

Nel caso di calcolare la matrice risultante C concorrentemente in uno spazio sovra-allocato, è necessario effettuare anche una operazione di copia finale, per riordinare le righe o raccoglierne i non zero prodotti nell output dell'operazione di SpMM.

2.3.2 Prodotto Simbolico Accurato

L'operazione di determinare esattamente la dimensione della matrice prodotta da SpMM è denominata in vari articoli Prodotto Simbolico e la seguente operazione di moltiplicazione dei vari elementi non zero è denominata Prodotto Numerico.

Un approccio tipico per determinare la dimensione di una riga risultante al operazione di SpMM, è quella di mantenere in una struttura di ricerca gli indici corrispondenti ai non zeri risultanti alle operazioni di prodotto, senza effettuare le costose operazioni di moltiplicazione floating point.

2.4 Algoritmi

Molte delle ricerche riguardo SpMM sono basate sull'algoritmo di Gustavson [Gus78], un algoritmo sequenziale basato sulla formulazione Row-by-row, riportate in forma

di pseudo-codice in 2.4 e graficamente in 2.5.

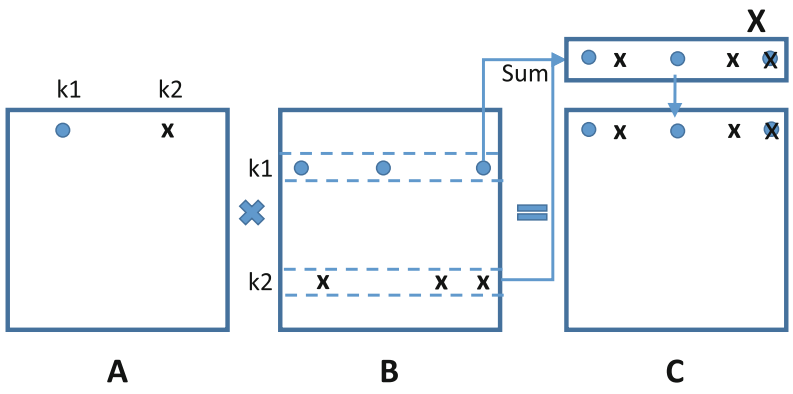
FIGURA 2.4: pseudo-codice dell'algoritmo di Gustavson

Algorithm 1: Pseudocode for the Gustavson's SpGEMM algorithm.

Input: $A \in \mathbb{R}^{p \times q}, B \in \mathbb{R}^{q \times r}$
Output: $C \in \mathbb{R}^{p \times r}$

```
for a_i* ∈ A do
    for a_ij ∈ a_i* and a_ij is non-zero do
        for b_jk ∈ b_j* and b_jk is non-zero do
            value ← a_ij * b_jk;
            if c_ik ∉ c_i* then
                c_ik ← 0;
            end
            c_ik ← c_ik + value;
        end
    end
end
```

FIGURA 2.5: rappresentazione grafica di una iterazione dell'algoritmo di Gustavson



È possibile notare come nell'algoritmo vi sia l'esigenza di accumulare le righe risultanti della matrice C. Meccanismi efficienti per realizzare questa funzionalità sono analizzati successivamente in 2.5.3.1

2.5 Algoritmi paralleli con formulazione multidimensionale

Gli algoritmi paralleli risolutivi del prodotto tra matrici sparse possono essere classificati in base al partizionamento del carico di lavoro sui processi.

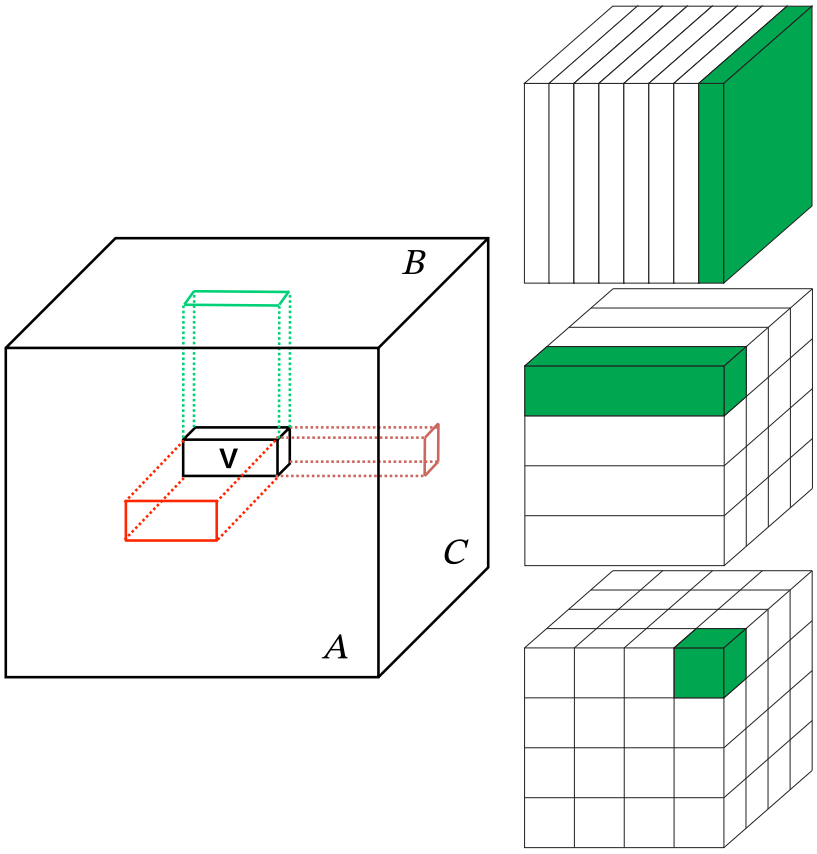


FIGURA 2.6: Rappresentazione grafica dell'assegnamento dei task per la risoluzione di SpMM in 1,2 e 3 dimensioni, evidenziando rispettivamente dall'alto: un layer, un Fiber ed un Cuboid

Una visualizzazione grafica delle operazioni di SpMM, partizionabili tra delle unità di esecuzione è data dal cubo di lavoro W (in figura 2.6) in cui ogni moltiplicazione di elementi non zero $a_{i,k} * b_{k,j}$ (minima operazione assegnabile) è rappresentabile con voxel $W(i, j, k)$ [DA20].

I sottoinsiemi dei voxel di W ottenuti fissando uno o due indici sono denominati:

- Layers: W(i,:,:),W(:,j,:),W(:,:,k)
- Fibers: Intersezioni di Layers relativi a indici diversi, e.g. W(i,j,:)

- Cuboid: Sottoinsiemi di W con tutte le dimensioni minori delle dimensioni delle matrici corrispondenti

È possibile dare una rappresentazione grafica del partizionamento del carico di lavoro per SpMM mediante gli elementi appena citati.

2.5.1 Matrici ipersparse e rappresentazione DCSC

È possibile definire una matrice sparsa A, come **ipersparsa** se $nnz(A)$ è inferiore alla sua dimensione più grande N [BG08]
Tipicamente questa tipologia di matrici è rara nell'algebra lineare numerica. Infatti, nella soluzione di sistemi lineari, le matrici sono quadrate, ed avere un numero di nonzeri inferiore alla dimensione vorrebbe dire che alcune equazioni non hanno coefficienti, il che nei problemi provenienti da equazioni differenziali chiaramente non ha senso. Tuttavia, questa tipologia di matrici trova applicazioni nel partizionamento di matrici sparse e nel processamento grafi, particolarmente se in parallelo.

Data una suddivisone bidimensionale di una matrice sparsa per un suo processamento parallelo in una griglia di p processi, si ha che ogni processo verrà assegnato ad una matrice di dimensione circa $(n/\sqrt{p}) \times (n/\sqrt{p})$.
L'occupazione globale di spazio di memoria sarà pari a $O(nnz + p \cdot n/\sqrt{p}) = O(nnz + n \cdot \sqrt{p})$ con CSC che è maggiore dell'occupazione totale della matrice in un singolo processo $O(n + nnz)$.
Allo scalare del numero di processi p, il termine $n\sqrt{p}$ domina nnz.

Per queste ragioni è possibile utilizzare la rappresentazione **D**ouble**C**ompressed**S**parse**C**olumns (visualizzata in figura 2.7) per partizionamenti spinti di matrici sparse, dato che ha un'occupazione di memoria pari ad $O(nnz)$ eliminando eventuali ripetizioni nell'array di puntatori delle colonne (JC) nel formato CSC.
Inoltre, è possibile favorire un rapido accesso alle colonne della matrice mediante un array ausiliario di indici delle colonne non zero.

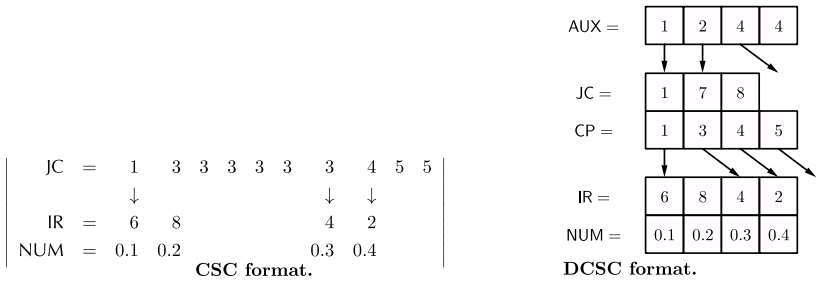


FIGURA 2.7: confronto delle rappresentazioni DCSC e CSC

2.5.2 Moltiplicazione tra matrici ipersparse DCSC

Segue la descrizione di un algoritmo risolutivo per la moltiplicazione tra matrici ipersparse [BG08], basato su una formulazione outer-product ed utilizzato in alcuni algoritmi SpMM.
lo pseudo-codice dell'algoritmo è riportato in 2.8

Hypersparse matrix multiply
Pseudocode for hypersparse matrix matrix multiplication algorithm.

$\mathbf{C} : \mathbb{R}^{S(M\times N)} = \text{HYPERSPARSEGEMM}(\mathbf{A} : \mathbb{R}^{S(M\times K)}, \mathbf{B^T} : \mathbb{R}^{S(N\times K)})$

```
1  isect ← INTERSECTION(A.JC, B^T.JC)
2  for j ← 1 to |isect|
3        do CARTMULT-INSERT(A, B^T, PQ, isect, j)
4           INCREMENT-LIST(isect, j)
5  while ISNOTFINISHED(isect)
6        do (key, value) ← EXTRACT-MIN(PQ)
7           (product, i) ← UNPAIR(value)
8           if key ≠ TOP(Q)
9              then ENQUEUE(Q, key, product)
10             else  UPDATETOP(Q, product)
11          if ISNOTEMPTY(isect(i))
12             then CARTMULT-INSERT(A, B^T, PQ, lists, isect, i)
13                  INCREMENT-LIST(isect, i)
14 CONSTRUCT-DCSC(Q)
```

FIGURA 2.8: pseudocodice dell'algoritmo sequenziale hypersparseMM,per SpMM tra matrici ipersparse

Fase di preprocessamento
viene effettuata una trasposizione di B, così da avere un indicizzamento rapido delle righe di B in formato DCSC, necessario per la formulazione outer-product.
Viene effettuata l'intersezione tra gli indici delle colonne non zero di A e delle righe non zero di B, identificando così il set $Isect = A.JC \cap B^T.JC$ degli indici che partecipano all'outer-product.

Successivamente vengono effettuati $|Isect|$ prodotti cartesiani 2.9 generanti matrici $a_{*i} \cdot b_{i*}$, i cui elementi è possibile mettere in biiezione con la lista di indici (r_{id}, c_{id}) derivante dal prodotto cartesiano tra gli indici di riga non zero della colonna i-esima di B^T e gli indici di riga non zero della colonna i-esima di A.
La matrice risultante C può essere ottenuta mediante l'unione di queste liste, sommando i contributi degli elementi aventi gli stessi indici in liste diverse.
Per implementare la costruzione di C dagli outer-products è utilizzata una coda con priorità contenete elementi relativi ai prodotti cartesiani per le colonne di A e le colonne di $B^T \in Isect$, aventi per chiave (r_{id}, c_{id}) e per valore il prodotto relativo e l'indice della colonna di A o B^T relativa.
Verrà ripetutamente estratto il minimo dalla coda e reinserito l'elemento successivo dalla lista di indici di relativa, sommando i contributi di elementi con la stessa chiave. I risultati verranno accumulati in uno stack mediante una rappresentazione matriciale a coordinate degli elementi per poi essere convertiti nella matrice finale C in formato DCSC.

La complessità computazionale dell'algoritmo è $O(nzc(A) + nzr(B) + flops \cdot lg(ni))$ dove ni è la dimensione della coda con priorità e $flops$ è il numero di operazioni aritmetiche necessarie per il prodotto di A e B.

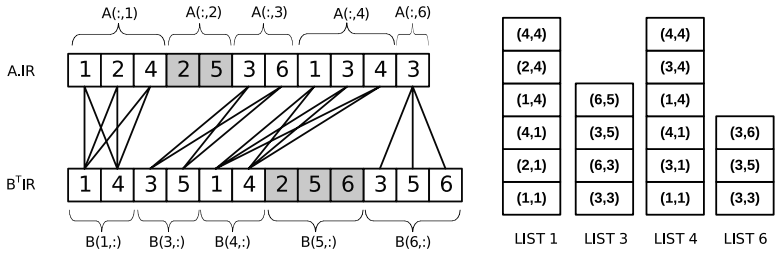


Figure 14.8. Cartesian product and the multiway merging analogy.

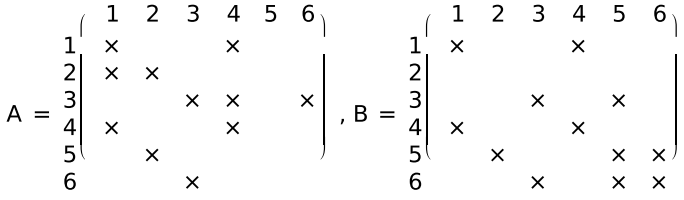


Figure 14.9. Nonzero structures of operands A and B.

FIGURA 2.9: Rappresentazione grafica del prodotto cartesiano a supporto dell'algoritmo hypersparseMM

2.5.3 Algoritmi con un partizionamento 2D

2.5.3.1 Derivati di Gustavson

Una parallelizzazione della formulazione per righe dell'algoritmo di Gustavson è descritta in [MMAPD08], sfruttando la rappresentazione di matrici sparse in formato CSR. Lo pseudocodice dell'algoritmo è riportato assieme ad una sua rappresentazione grafica in figura 2.10
Viene effettuato un partizionamento delle righe di A e delle colonne di B e coppie di partizioni corrispondenti vengono utilizzate per calcolare un blocco bidimensionale della matrice risultante C.

In accordo con la formulazione per righe dell'algoritmo di Gustavson, vengono accumulati i contributi degli elementi non zero delle righe di A e corrispettive porzioni delle righe di B relative al calcolo di un blocco di C. Per effettuare efficientemente questa operazione, i vettori sparsi risultanti dalle iterazioni dell'algoritmo, vengono accumulati in vettore denso X, semplicemente sommandone le componenti non zero nelle relative locazioni di X.

In questo contesto, una soluzione di accumulazione alternativa ad X potrebbe essere quella di accumulare i contributi per una riga di C utilizzando una hash-table di elementi aventi per chiave l'indice di colonna. Tuttavia, quest'approccio ha lo svantaggio di avere l'overhead relativo alla computazione delle funzioni hash e gestione di eventuali liste di collisione.
Al termine del calcolo di una riga di un blocco di C, l'accumulare X viene convertito in una riga CSR con l'ausilio di un ulteriore vettore di indici di elementi non zero sommati in X.

Utilizzando un partizionamento delle colonne di B è possibile partizionare anche l'accumulatore X relativo ad una riga risultante di C. In questa maniera si riduce il numero di cache line toccate dagli aggiornamenti di X, riducendo il numero di cache miss relativi al ciclo interno dell'algoritmo.
Gli autori dell'algoritmo hanno verificato la riduzione di cache miss utilizzando degli hardware counter per verificare il numero di cache miss L2 (tipicamente catturati dalla cache LLC) al variare del partizionamento di B

Tuttavia il partizionamento di B in rappresentazioni CSR separate, oltre che comportare un overhead computazionale, può comportare svantaggi in termini di banda di memoria durante la lettura dei blocchi di B, che possono essere notevolmente più sparsi della matrice originaria.

Un beneficio dal partizionamento di B e di X, contro gli overhead citati, è presente quando si ha certezza che per una significativa porzione delle righe risultanti di C, l'occupazione dei non zeri, corrispondenti agli aggiornamenti dell'accumulatore X, è maggiore della dimensione della cache L2.
Per quantificare il numero di non zero nelle righe risultanti di C viene utilizzata una metrica basata sulla stima di non zero in $c_{i*} = e\_nnz(i)$ di [MMAPD08] dove si partiziona B se $e\_nnz = \frac{\sum_{i:e\_nnz(i)>L2\_FP\_WORDS} e\_nnz(i)}{\sum_{i=1}^{m} e\_nnz(i)} > 0.3$

```
1 part = Partition(A, B)
2 #pragma omp parallel for
  schedule(dynamic)
3 forall the p in part do
4     for all rows A_{i,:} in p do
5         Reset X to 0
6         for each nonzero A_{i,j} in A_{i,:} do
7             Load partition B_{j,cols(p)} in p
8             X += A_{i,j} · B_{j,cols(p)}
9         Sparsify X to C_{i,cols(p)}
```

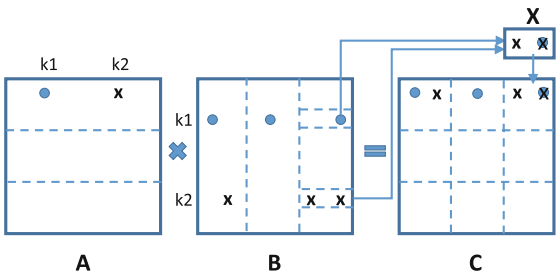


FIGURA 2.10: adattamento parallelo dell'algoritmo di Gustavson a sinistra ed una sua rappresentazione grafica a destra

L'uso di un vettore denso per operazioni tra matrici sparse è stato affrontato originariamente da [JRGS91].

2.5.3.2 Dense SUMMA

Un partizionamento bidimensionale della risoluzione del prodotto tra matrici dense è realizzato dell'algoritmo parallelo SUMMA [GW97] che è alla base di una sua controparte per matrici sparse [BG12].
Segue una descrizione del algoritmo dense SUMMA.

Il calcolo di $C = AB$ è suddiviso in sottomatrici assegnate a processi organizzati in una griglia bidimensionale di dimensione $p_r \times p_c$.

È possibile calcolare un blocco della matrice C come: $C_{ij} = \overbrace{(A_{i1}|\ldots|A_{ip_c})}^{\tilde{A}^{i*}} \cdot \overbrace{\begin{pmatrix} B_{1j} \\ \vdots \\ B_{p_r j} \end{pmatrix}}^{\tilde{B}^{*j}}$

dove si usa la notazione:

- $\tilde{a}^{i*}_{*l} \in \tilde{A}^{i*}$ rappresenta la colonna l-esima nella riga i-esima nella decomposizione a blocchi di A

- $\tilde{b}^{*j}_{l*} \in \tilde{B}^{*j}$ rappresenta la riga l-esima nella colonna j-esima nella decomposizione a blocchi di B.

Applicando la formulazione outer-product su $\tilde{A}^{i*}$ e $\tilde{B}^{*j}$ si ha che $C_{ij} = \sum_{l=1}^{k} \tilde{a}^{i*}_{*l} \cdot \tilde{b}^{*j}_{l*}$
Considerando un partizionamento delle matrici in sottomatrici all'interno della griglia di processi si può eseguire l'operazione di SpMM in parallelo come riportato dallo pseudocodice 2.11.

$C_{ij} = 0$
for $l = 0, k-1$
broadcast $\tilde{a}^l_i$ within my row
broadcast $\tilde{b}^j_l$ within my column
$C_{ij} = C_{ij} + \tilde{a}^l_i \tilde{b}^{j\,T}_l$
endfor

FIGURA 2.11: pseudocodice dell'algoritmo SUMMA relativalmente al processo P_{ij}

Nell'articolo originale vengono discussi miglioramenti a questa formulazione basati su uno scambio di dati tra i processi in un sistema a memoria distribuita mediante una comunicazione ad anello ed una riformulazione del calcolo di C_{ij} utilizzando blocchi di colonne di $\tilde{A}^{i*}$ e righe di $\tilde{B}^{*j}$.

2.5.3.3 Sparse SUMMA

Un partizionamento bidimensionale del problema SpMM è realizzato dall'algoritmo Sparse SUMMA [BG12], derivato dalla sua controparte per matrici dense.
È riportato lo pseudo-codice dell'algoritmo sparseSUMMA in 2.12 e una sua iterazione graficamente in 2.13.
Le matrici sono divise in blocchi e rappresentante in formato DCSC, trasponendo la matrice B per avere un indicizzamento rapido delle sue righe.
Procedendo in blocchi di colonne di una sottomatrice di A e della corrispettiva sottomatrice di B^T, lungo la dimensione comune di A e B, vengono calcolati in parallelo le sottomatrici C_{ij}, utilizzando l'algoritmo hypersparseMM 2.8.
Il costo computazionale dell'algoritmo è $O(\frac{dn}{\sqrt{p}} + \frac{d^2n}{p} lg(\frac{d^2n}{p}))$

Algorithm 2 Operation $\mathbf{C} \leftarrow \mathbf{AB}$ using Sparse SUMMA

Input: $\mathbf{A} \in \mathbb{S}^{m \times k}, \mathbf{B} \in \mathbb{S}^{k \times n}$: sparse matrices distributed on a $p_r \times p_c$ processor grid
Output: $\mathbf{C} \in \mathbb{S}^{m \times n}$: the product $\mathbf{AB}$, similarly distributed.

1: **procedure** SPARSESUMMA($\mathbf{A}, \mathbf{B}, \mathbf{C}$)
2: **for** all processors $P(i, j)$ **in parallel do**
3: $\mathbf{B}_{ij} \leftarrow (\mathbf{B}_{ij})^\mathsf{T}$
4: **for** $q = 1$ to k/b **do** ▷ blocking parameter b evenly divides k/p_r and k/p_c
5: $c = (q \cdot b)/p_c$ ▷ c is the broadcasting processor column
6: $r = (q \cdot b)/p_r$ ▷ r is the broadcasting processor row
7: $lcols = (q \cdot b) \bmod p_c : ((q+1) \cdot b) \bmod p_c$ ▷ local column range
8: $lrows = (q \cdot b) \bmod p_r : ((q+1) \cdot b) \bmod p_r$ ▷ local row range
9: $\mathbf{A}^{rem} \leftarrow$ BROADCAST($\mathbf{A}_{ic}(:, lcols), P(i, :)$)
10: $\mathbf{B}^{rem} \leftarrow$ BROADCAST($\mathbf{B}_{rj}(:, lrows), P(:, j)$)
11: $\mathbf{C}_{ij} \leftarrow \mathbf{C}_{ij} +$ HYPERSPARSEGEMM($\mathbf{A}^{rem}, \mathbf{B}^{rem}$)
12: $\mathbf{B}_{ij} \leftarrow (\mathbf{B}_{ij})^\mathsf{T}$ ▷ Restore the original $\mathbf{B}$

FIGURA 2.12: pseudocodice dell'algoritmo SparseSUMMA

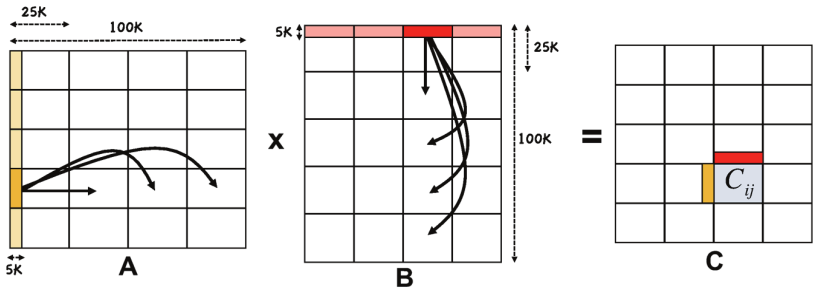


FIGURA 2.13: una iterazione dell'algoritmo sparseSUMMA

2.5.4 Algoritmi con un partizionamento 3D

Un partizionamento tridimensionale del problema SpMM è realizzato dall' algoritmo Split-3D-SpMM [AAW16], dove al partizionamento bidimensionale descritto precedentemente, viene aggiunto una ulteriore suddivisione delle sottomatrici in blocchi nella terza dimensione della griglia di processi. Il partizionamento delle matrici è rappresentabile sul cubo di lavoro W come riportato in figura 2.14, dove il processo P(i,j,k) possiede la porzione della matrice di A:
$A(im/p_r : (i+1)m/p_r - 1, jn/p_c + kn/(p_c p_l) : jn/p_c + (k+1)n/(p_c p_l) - 1)$

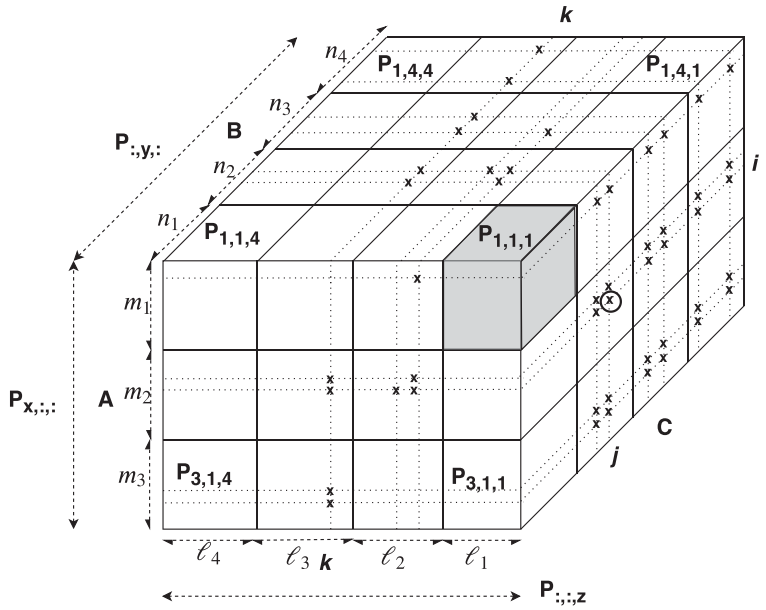


FIGURA 2.14: rappresentazione 3D della suddivisione delle operazioni per la computazione di SpMM

L'algoritmo è riportato nello pseudo-codice 2.15 e una sua iterazione è raffigurata graficamente in 2.16.

In maniera simile al caso di sparseSUMMA si procede in parallelo calcolando il prodotto di sotto blocchi corrispondenti alle sottomatrici di A e B, accumulando risultati intermedi la cui collocazione è relativa all'intera fiber W(i,j,:).
Infine i risultati intermedi dei processi P(i,j,:), vengono distribuiti lungo la fiber W(i,j,:), sommando i contributi relativi agli stessi indici.

Algorithm 2 Operation $\mathbf{C} \leftarrow \mathbf{AB}$ using Split-3D-SpGEMM.

Input: $\mathbf{A} \in \mathbb{S}^{m \times l}, \mathbf{B} \in \mathbb{S}^{l \times n}$: matrices on a $\sqrt{p/c} \times \sqrt{p/c} \times c$ process grid
Output: $\mathbf{C} \in \mathbb{S}^{m \times n}$: the product $\mathbf{AB}$, similarly distributed

1: **procedure** Split-3D-SpGEMM($\mathbf{A}, \mathbf{B}, \mathbf{C}$)
2: $locinndim = l/\sqrt{pc}$ ▷ inner dimension of local submatrices
3: **for** all processes $P(i, j, k)$ **in parallel** **do**
4: $\hat{\mathbf{B}}_{ijk} \leftarrow$ AllToAll($\mathbf{B}_{ij:}, P(i, j, :)$) ▷ redistribution of $\mathbf{B}$ across layers
5: **for** $r = 1$ to $\sqrt{p/c}$ **do** ▷ r is the broadcasting process column and row
6: **for** $q = 1$ to $locinndim\,/b$ **do** ▷ b evenly divides $locinndim$
7: $locindices = (q-1)b : qb - 1$
8: $\mathbf{A}^{rem} \leftarrow$ Broadcast($\mathbf{A}_{irk}(:, locindices), P(i, :, k)$)
9: $\mathbf{B}^{rem} \leftarrow$ Broadcast($\hat{\mathbf{B}}_{rjk}(locindices, :), P(:, j, k)$)
10: $\mathbf{C}^{int}_{ij:} \leftarrow \mathbf{C}^{int}_{ij:} +$ HeapSpGEMM($\mathbf{A}^{rem}, \mathbf{B}^{rem}$)
11: $\mathbf{C}^{int}_{ijk} \leftarrow$ AllToAll($\mathbf{C}^{int}_{ij:}, P(i, j, :)$)
12: $\mathbf{C}_{ijk} \leftarrow$ LocalMerge($\mathbf{C}^{int}_{ijk}$)

FIGURA 2.15: pseudocodice dell'algoritmo Split3DSpMM, per una risoluzione parallela di SpMM con un partizionamento 3D nel caso semplificato di una griglia di processi $\sqrt{p/c} \times \sqrt{p/c}\ x\ c$

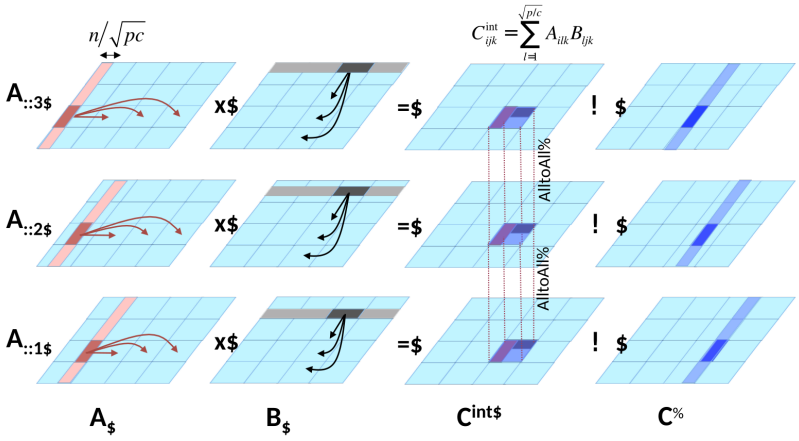


FIGURA 2.16: una iterazione dell'algoritmo Split3DSpMM

2.6 Algoritmi paralleli basati su formulazione Outer-Product

Implementazioni di SpMM basate su formulazioni di Outer-Product, godono di una potenziale migliore località spaziale dei non zero acceduti durante il prodotto, come spiegato in [ZG20].
Considerando rispettivamente formulazioni row-by-row e col-by-col per SpMM e denominando la matrice dominante nei prodotti rispettivamente A e B, si ha una buona località spaziale per le matrici dominanti e risultanti, ma non per quelle non dominanti.
Questo è dovuto al fatto che la matrice non dominante deve essere acceduta per righe o colonne in un ordine derivato dal pattern di sparsità dei non zeri delle matrici dominanti, che è assimilabile ad un ordine casuale.

Nel caso row-by-row ad esempio, si può avere un cattivo uso della cache in scenari dove:
una riga di B deve essere riletta varie volte, ma l'ordinamento degli accessi alle righe di B causa che, il più delle volte, la riga non sia in cache quando necessario.
In questi casi si ha che formulazioni row-by-row o col-by-col causano uno spreco di banda di memoria, il che è particolarmente critico per un problema memory bound come SpMM.

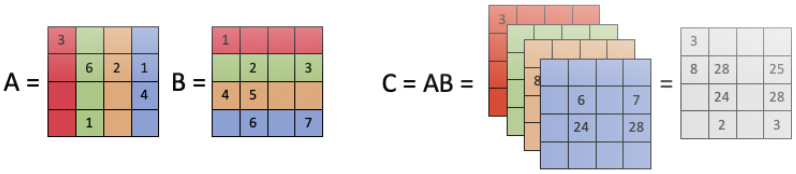


FIGURA 2.17: Visualizzazione di un prodotto tra matrici 4x4 mediante Outer-Product

2.6.1 ESC - PropagationBlocking

Una soluzione per l'operazione di SpMM mediante Outer-Product è riportata in [ZG20], dove viene usato un algoritmo basato sul generale approccio risolutivo Expand-Sort-Compress, applicato ad una formulazione Outer-Product per SpMM.

Le principali fasi dell'algoritmo sono:

- $\hat{C} \leftarrow$ Symbolic(A,B)
  Allocazione dei risultati intermedi al prodotto mediante un UpperBound

- $\hat{C} \leftarrow$ Expand(A,B)
  Esecuzione dei prodotti a rango 1 della formulazione Outer-Product in formato COO

- Sort($\hat{C}$)
  Ordinamento mediante radix-sort dei prodotti intermedi mediante le loro coordinate risultanti

- $C \leftarrow$ Compress($\hat{C}$)
  unione dei risultati intermedi, sommandoli nella matrice finale

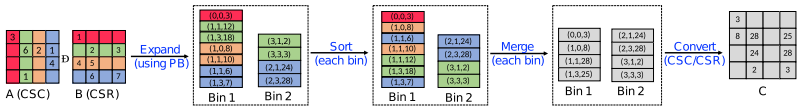


FIGURA 2.18: Rappresentazione grafica degli step principali dell'algoritmo

---

2.6.1.1 Symbolic – UB estimate

Analogamente a quanto descritto precedentemente in 2.3, preferendo una preallocazione delle strutture intermedie necessarie rispetto ad effettuare (ri)allocazioni dinamiche, viene calcolata la dimensione dei risultati intermedi che corrisponde ad un UpperBound della dimensione finale del risultato.
Seguendo un approccio Outer-Product per SpMM, il numero di non zero al termine della fase Expand è pari ad: $\sum_{i=1}^{k} |I^i(A)| * |I_i(B)|$

2.6.1.2 Expand

In questa fase vengono effettuati i prodotti richiesti dalla formulazione Outer-Product, salvando i non zeri come tuple composte dal risultato numerico e le coordinate di righe e colonne (formato COO).
Per questioni di efficienza è necessario che la matrice A sia in un formato con colonne contigue (CSC) e B in un formato con righe contigue (CSR).
Queste tuple vengono generate e raggruppate in parallelo in bin in base agl'indici di riga dei corrispondenti elementi di A.
Per una maggiore efficienza ogni thread mantiene una copia locale di tutti i m bin possibili, effettuando periodicamente una operazione di *flush* dei dati locali nei bin globali.

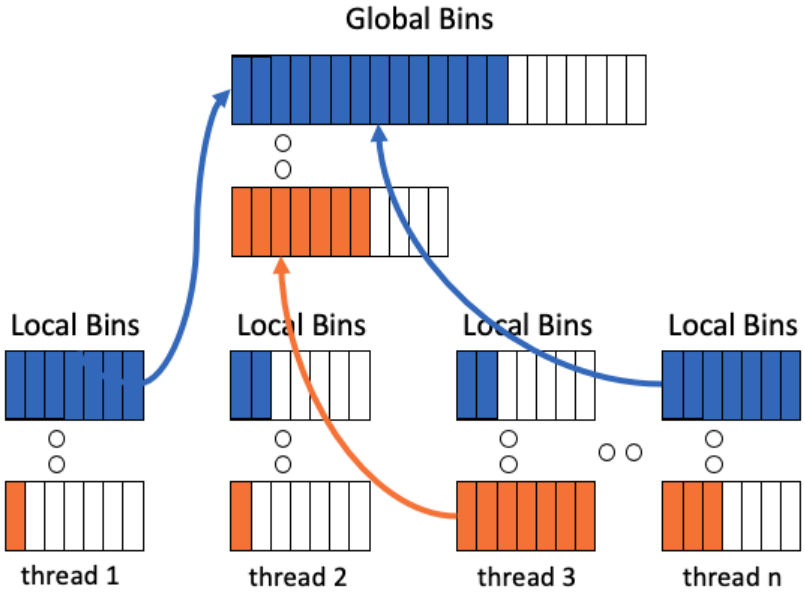


FIGURA 2.19: Rappresentazione grafica delle operazioni di flush periodiche dei bin locali ai thread nei bin globali

Il termine "PropagationBlocking" deriva da questa organizzazione dei prodotti in Bin locali e globali, consentendo insieme alle fasi successive una somma efficiente dei valori corrispondenti allo stesso elemento non zero nella matrice risultante.

2.6.1.3 Sort

Si effettua un sorting delle tuple prodotte in $\hat{C}$ nei bin globali nella fase precedente, ordinandole in base ai campi relativi alle coordinate di righe e colonne.
Gli autori dell'articolo asseriscono di usare una versione inplace di radix-sort, che raggruppa le chiavi in base alla posizione del byte più significativo (richiedendo cosi un numero di passate sui dati proporzionale al numero di byte nelle chiavi), dando performance migliori rispetto ad algoritmi basati su comparazioni per chiavi piccole.
Utilizzando un numero di bin tale da averene la maggior parte in dimensione tale da poter entrare in cache L2 o L3 è possibile effettuare il sorting in cache, attenuando il costo dovuto alle riletture dei dati per l'operazione di ordinamento.

2.6.1.4 Compress

Si sommano tutte le tuple aventi stesse cordinate di riga e colonna, corrispondenti allo stesso elemento non zero nella matrice risultante.
Grazie all'ordinamento effettuato nella fase precedente, è possibile completare questa

fase con una singola scansione di tutti i Bin globali.

Per facilitare le fasi di Sort e Compress, il numero di bin globali può essere determinato nella fase simbolica dividendo il numero di tuple attese per la dimensione della cache L2 del dispositivo di calcolo. Questo consente che i bin globali nelle ultime 2 fasi possano entrare in cache L2, migliorando l'utilizzo della memoria durante l'esecuzione in parallelo.

CAPITOLO 3

IMPLEMENTAZIONI OPENMP: PRODOTTO SIMBOLICO

In questa sezione si descrivono alcune implementazioni che ho realizzato per il calcolo del numero di non zeri del prodotto tra matrici sparse, introdotto precedentemente in 2.3.
In letteratura è consueto utilizzare la terminologia di prodotto simbolico per indicare il calcolo esatto della dimensione della matrice risultante all'operazione di SpMM.
Nel seguito si indicherà con prodotto simbolico, la generale operazione di determinare, con un bound o con precisione, la dimensione del Prodotto tra matrici sparse.
Come verrà descritto nel dettaglio in 5.1, al fine di avere la massima efficienza nella generazioni di multiple implementazioni dello stesso codice a compile-time, per supportare l'integrazione di tutte le implementazioni descritte in un progetto fortran come AMG4PSBLAS `https://github.com/sfilippone/amg4psblas`, saranno presenti nei frammenti di codice successivi macro come `CAT OFF_F`, necessarie ad ottenere molteplici versione del codice a tempo di pre processamento.

In base al tipo di implementazione del prodotto numerico e di partizionamento dei dati tra i thread, è necessaria una fase simbolica di SpMM per calcolare il numero di non zeri a livello di:

1. intere matrici C e $\hat{C}$
2. righe di C e $\hat{C}$
3. partizioni di colonne in righe di C e $\hat{C}$

Il livello di output prodotto da ogni livello può essere inclusivo di quello prodotto dal livello inferiore.
La necessità di avere una predizione della dimensione del risultato con un dettaglio differente è dovuta al tipo di implementazione usata per il Prodotto Numerico e sarà descritta successivamente in 4.2.1.

La numerazione appena introdotta per il livello di dettaglio dell'output del prodotto simbolico sarà riferita nel seguito.

3.1 UpperBound

Come descritto in 2.3.1, la determinazione di un limite superiore al numero di non zeri del risultato di SpMM è computabile rapidamente e semplicemente.

Nel caso di dover determinare un prodotto simbolico a livello 1 o 2 si ha un costo computazionale proporzionale al numero di non zeri di A.

Nel caso di dover calcolare un bound per ogni partizione di colonne di ogni riga di C e $\hat{C}$, per implementazioni della fase numerica con partizionamento bidimensionale delle matrici, ho utilizzato il seguente approccio.
Un bound superiore della dimensione della k-esima partizione di colonne di c_{i*} è calcolabile restringendo la formula $|I_i(C)| \leq \sum_{j \in I_i(A)} |I_j(B)|$ all' sottoinsieme di elementi non zero relativi della k-esima partizione di colonne di B.
Segue l'implementazione di questa operazione.

```
1 /*
2  * return matrix @A.M x @gridCols of upper bounded num of non zeros
3  * for each of the @gridCols col partitions of the output matrix AB = @A * @B rows
4  * also appended at the end for the cumulative total size of the matrix AB
5  */
6 inline idx_t* CAT(spMMSizeUpperboundColParts_,OFF_F)
7   (spmat* A,spmat* B,ushort gridCols,idx_t* bColPartOffsets){
8     idx_t* rowPartsSizes = calloc((A->M*gridCols +1),  sizeof(*rowPartsSizes));
9     if (!rowPartsSizes){
10         ERRPRINT("rowPartsSizes calloc errd\n");
11         return NULL;
12     }
13
14     idx_t fullMatBound = 0;
15     #pragma omp parallel for schedule(static) reduction(+:fullMatBound)
16     for (idx_t r=0;  r<A->M; r++){
17         //for each A.row -> sum B.colParts lens
18         for (idx_t jj=A->IRP[r]-OFF_F,j,rlen;  jj<A->IRP[r+1]-OFF_F; jj++){
19             j = A->JA[jj] - OFF_F;
20             for (idx_t gc=0,bPartID=IDX2D(j,gc,gridCols);  gc < gridCols; gc++,bPartID++){
21                 rlen = bColPartOffsets[bPartID+1] - bColPartOffsets[bPartID];
22                 rowPartsSizes[ IDX2D(r,gc,gridCols) ]   += rlen;
23                 fullMatBound    += rlen;
24             }
25         }
26     }
27     rowPartsSizes[ A->M*gridCols ]  = fullMatBound;
28     return rowPartsSizes;
29 }
```

Nella funzione precedente viene sfruttato un partizionamento 2D di B mediante una matrice di offset, `bColPartOffsets`, dove l'elemento i, j è l'indice dell'elemento non zero iniziale della j-esima partizione di colonne della i-esima riga. Questa struttura di supporto è descritta nel dettaglio insieme ad altre metodologie di partizionamento per matrici sparse in 5.7.
Il bound sulla `gc`-esima partizione della `r`-esima riga di C è calcolato a riga 21, come

precedentemente descritto.
Per ogni livello di dettaglio del calcolo del prodotto simbolico, viene calcolata anche la dimensione totale della matrice C, mediante la riduzione offerta da OpenMp con la clausola `reduction(+:..)`.
In generale l'overhead relativo ai vari componenti di OpenMp da istanziare per effettuare solamente una operazione di riduzione possono precludere un beneficio prestazionale per input piccoli.
Tuttavia, l'uso di questa clausola all'interno di un ciclo parallelizzato più ampio può ammortizzare l'overhead di inizializzazione di OpenMp, dando un beneficio di performance.

3.2 Calcolo Accurato

Per il prodotto simbolico accurato, focalizzandomi su algoritmi derivati da quello di Gustavson [Gus78] con formulazione row-by-row, ho realizzato implementazioni per tutti i livelli di dettaglio descritti precedentemente in 3.
Il Prodotto Simbolico accurato tra 2 matrici $C = A * B$ è realizzato seguendo l'approccio row-by-row: mantenendo il conto, senza ripetizioni, degli indici di colonna degli elementi non zero delle righe di B relativi agli indici di colonna di una riga di A.
Per la gestione degli indici, mi sono basato principalmente su due strutture ausiliarie per mantenere gli indici degli elementi non zero risultanti dall'operazione di SpMM:

- RedBlack trees, con un nodo per elemento risultante non zero , descritti in 3.2.2
- dei set indicizzati di flag, con un elemento `true` per elemento risultante non zero , sottoforma di
  - un insieme di bitmap
  - un array di byte

  descritti in 3.2.3

Per ogni funzione esportata dal modulo del Prodotto Simbolico accurato è presente una variabile per scegliere la versione dell'implementazione con la struttura ausiliaria di ricerca desiderata.

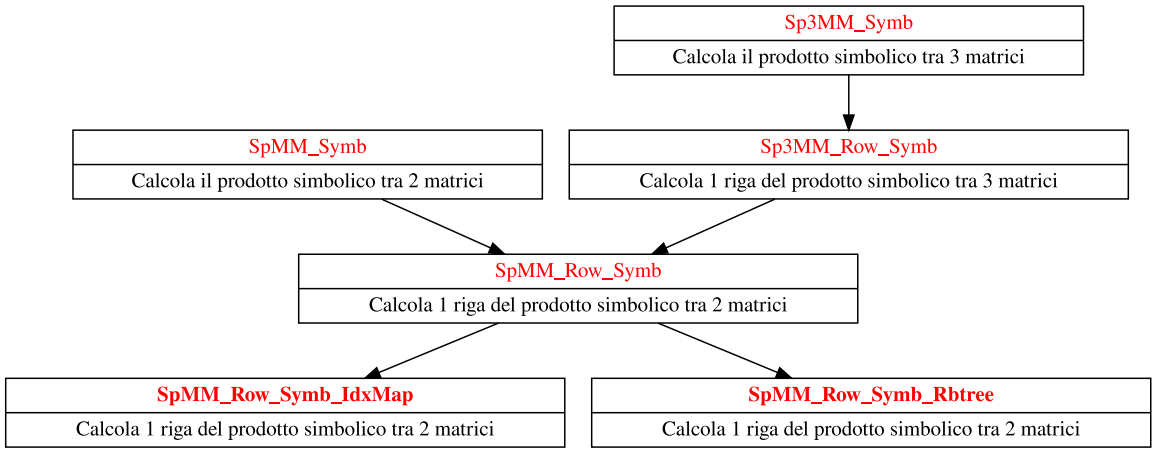
3.2.1 Strutturazione delle funzioni per supportare diverse implementazioni di Prodotto Numerico

Computare il Prodotto Simbolico in parallelo necessita in primis di effettuare una allocazione iniziale delle varie strutture di supporto al fine di evitare allocazioni

dinamiche come precedentemente descritto in 2.3.
In seguito, è possibile determinare concorrentemente dei sotto insiemi del Prodotto Simbolico, con un livello di dettaglio di output adeguato alle esigenze dell'implementazione del Prodotto Numerico scelta.

Dato l'obiettivo di supportare implementazioni con formulazione row-by-row, ho deciso di usare il calcolo del Prodotto Simbolico di una singola riga della matrice risultante come elemento base di parallelismo ed operazione da assegnare ai singoli thread.
Conseguentemente, la funzione che realizza questa operazione è alla base di tutte le funzioni più avanzate per realizzare il Prodotto Simbolico per la moltiplicazione tra 2 o 3 matrici.
Di seguito viene rappresentata schematicamente la strutturazione delle funzioni per ottenere tutte le implementazioni necessarie del problema in oggetto, dove una freccia direzionata tra 2 funzioni $a \rightarrow b$ indica che a chiama b.

FIGURA 3.1: Rappresentazione schematica delle dipendenze delle funzioni appartenenti al modulo del Prodotto Simbolico



È possibile notare come la funzione per determinare una riga del prodotto simbolico tra 2 matrici, `SpMM_Row_Symb_`, sia alla base di tutte le altre.

3.2.1.1 Generazione efficiente di versioni differenti di ogni funzione

Al fine di massimizzare l'efficienza delle implementazioni e il riuso del codice scritto per generare i vari livelli di output del prodotto simbolico (descritti in 3) ho associato alle funzione descritte in figura 3.1 versioni differenti, associate a delle funzionalità aggiuntive.

Di seguito le versioni alternative delle funzioni per eseguire la fase simbolica di SpMM e Sp3MM con specifici livelli di output,

- `OutIdxs_`: verranno ritornati, oltre che il numero di non zeri di ogni (partizione di) riga della matrice risultante, anche i relativi indici degli elementi non zero.

  Queste versioni sono necessarie per calcolare il triplo prodotto simbolico, come verrà descritto in 3.2.4.
  Inoltre, sono utili anche ad ottenere direttamente gli indici degli elementi non zero della matrice risultante (di cui i valori effettivi corrispondenti verranno computati nella fase numerica).

- `ColParts_` verranno ritornati, oltre che il numero di non zeri di ogni riga della matrice risultante, anche la loro distribuzione per ogni partizione di colonne di ogni riga.

  Queste versioni sono necessarie per eseguire SpMM con un partizionamento bidimensionale dei dati come verrà descritto in 4.3.2.

Le implementazioni aggiuntive appena descritte sono ottenute automaticamente a tempo di preprocessamento dalle funzioni base elencate in figura 3.1 mediante l'uso delle direttive del pre-processore C `#if` e molteplici `#include` dei file sorgenti [RMS21]. I dettagli a riguardo di questa tecnica saranno analizzati in 5.1.2.
Quest approccio implementativo può portare a diversi benefici, come:

- In casi in cui si debba traversare una struttura di indici di supporto è possibile, con una singola passata, sia copiare gli indici che tenerne traccia della loro distribuzione in partizioni separate
- Considerando un'altra soluzione, basata su una singola funzione che realizza le varie funzionalità richieste per il tipo di output necessario mediante delle condizioni su dei flag come approccio implementativo alternativo.
  Il codice prodotto dal mio approccio implementativo, in seguito alla fase di pre processamento, gode di:
  - assenza di tutte le extra istruzioni di branch richieste dall'approccio alternativo. In questo modo è potenzialmente più ottimizzabile in fase di compilazione.
  - l'uso di meno argomenti per chiamare alcune versioni generate
  - l'esclusione completa delle istruzioni non necessarie ad ogni versione generata.

Di seguito la realizzazione specifica delle varie versioni della calcolo di una riga del prodotto simbolico, ovvero la funzione `SpMM_Row_Symb_` vista precedentemnte nello schema 3.2.1, in base alla struttura di supporto usata per mantenere gli indici degli elementi non zero.

3.2.2 Uso di RedBlack tree

Usare in parallelo una implementazione di PriorityQueue è un approccio comune per gestire il mantenimento degli indici non zero per il Prodotto Simbolico, come fatto da [YN18].
Tra le varie possibilità disponibili, ho scelto di usare una implementazione di RedBlack tree, che ho ricavato personalmente dalla implementazione disponibile nel kernel Linux 5.10.85 (LTS), mediante un processo di porting in user-space. Il porting in userspace che ho realizzato dei RedBlack tree di linux è disponibile nella mia pagina: `https://github.com/andreadiiorio/redblackTree_linux_userspace` per altri usi.
Dettagli specifici dell'implementazione dei RedBlack tree e del loro processo di adattamento saranno trattati successivamente in 5.2.

Per realizzare la funzione target `SpMM_Row_Symb_` con i RedBlack tree in parallelo ho assegnato al thread in esecuzione: 1 riga di A (corrispondente alla riga target di C) ed un RedBlack tree allocato con un numero di nodi sufficiente per la riga più lunga di C, determinata con una riduzione sul risultato di un UpperBound del prodotto simbolico (visto in 3.1), con livello informativo dell'output 1 (rispetto alla numerazione introdotta in 3).
Durante la scansione degli indici i degli elementi non zero di B, corrispondenti agli elementi non zero di una riga di A, viene effettuato un inserimento di un nuovo nodo con chiave i nel RedBlack tree, se tale chiave non è stata già inserita. Ogni inserimento avvenuto incrementa il contatore degli elementi non zero della riga target di C.
Per implementare le versioni della funzione target che ritornino anche gli indici non zero effettivi della riga di C (`OutIdxs_`) e la loro distribuzione in partizioni di colonne (`ColParts_`) è sufficiente effettuare una singola scansione ordinata delle chiavi k inserite nel RedBlack tree prodotto, andando rispettivamente a salvare k e/o incrementare un contatore relativo alla partizione di colonne contente k.
Come già detto la scansione è singola e senza branch relativi a quale versione di `SpMM_Row_Symb_` si sta eseguendo grazie all'approccio di generazione automatica di diverse implementazioni mediante pre processore.
Nel caso di eseguire una versione della funzione in oggetto che ritorni gli indici degli elementi non zero della riga risultante, il salvataggio della chiave k del RedBlack tree generato, può avvenire in un array di supporto, allocato precedentemente alla funzione, o in alternativa può essere ordinato l'array di nodi RedBlack tree dato in input, *inplace*, contenente gli indici degli elementi non zero della riga target di C come chiavi, come è possibile vedere nella funzione successiva.

```
1 static inline idx_t CAT4(SpMM_Row_Symb_Rbtree,OUT_IDXS,COL_PARTS,OFF_F)
2   (
3    idx_t* aRowJA, idx_t aRowLen, spmat* b,rbRoot* root, rbNode* nodes
4    #if _OUT_IDXS  == TRUE && !defined OUT_IDXS_RBTREE_NODES
5    ,idx_t* outIdxs
6    #endif
```

```
7    #if _COL_PARTS == TRUE
8    ,ushort gridCols,idx_t* rowColPartsLens
9    #endif
10   )
```

Nel frammento di codice precedente è possibile vedere come la segnatura della funzione in analisi vari in base alla versione da realizzare. In particolare, il salvataggio degli indici non zero degli elementi della riga target di C (versione `OutIdxs_`), avviene sull'array di supporto `outIdxs` a meno che sia definita la macro di configurazione `OUT_IDXS_RBTREE_NODES`, che comporterà il riordinamento inplace del vettore di nodi RedBlack tree `nodes`, contente gli indici target come chiavi.

3.2.3 Uso di bitmap di indici o array di flag

Usare una struttura contente un insieme di flag indicanti la presenza di indici è un approccio alternativo ai RedBlack per realizzare la funzione target `SpMM_Row_Symb_` descritta in 3.2.1.1, potenzialmente con dei benefici prestazionali per alcune versioni.
Analogamente al caso dei RedBlack tree, viene assegnato al thread in esecuzione 1 riga di A, corrispondente alla riga target di C e un set di flag, inizializzati a false, di dimensione pari al numero di colonne N di C, sufficiente a contenere entries per ogni possibile indice della riga target di C.
Durante la scansione degli indici di colonna i degli elementi non zero di B corrispondenti a una riga di A, viene posto a true l'i-esimo flag e, se non era precedentemente posto false, viene anche incrementato il contatore degli elementi non zero della riga target di C.
La struttura contenete il set di flag relativi agli elementi non zero di una riga di C è definita nel tipo `nnz_idxs_flags_t` ed è configurabile a tempo di compilazione mediante la macro di configurazione `SPVECT_IDX_BITWISE`.

3.2.3.1 bitmap di indici

Se la macro di configurazione `SPVECT_IDX_BITWISE` è settata a `TRUE`, l'approccio usato sarà quello di salvare ogni indice di elementi non zero all'interno di un insieme di bitmaps.
Ogni indice da inserire sarà mappato in un bit specifico, da asserire ad 1, all'interno di una delle variabili del set di bitmaps. Nel caso peggiore il set di indici da inserire varia in $(0, N)$, dove N può essere molto grande.
Usando varibili di dimensione b bits è necessario allocarne al più $\lceil \frac{N}{b} \rceil$ per essere in grado di poter mappare l'intero set di indici.
Questo approccio è usato sia nel Prodotto Simbolico che Numerico e dettagli ulteriori sull'inserimento e controllo di indici nelle bitmaps saranno analizzati in 5.3.
Nel caso di dover realizzare efficientemente la versione `OutIdxs_` della funzione target(descritta in 3.2.1.1), è necessario utilizzare una struttura di supporto per salvare tutti gli indici degli elementi non zero che sono stati effettivamente inseriti in una bitmap. Per per ottemperare a questa necessità ho implementato 2 soluzioni,

selezionabili a tempo di compilazione con la macro di configurazione `IDX_RMUL_SYMB_RBTREE`.

- se `IDX_RMUL_SYMB_RBTREE` è configurata a `TRUE`, tutti gli indici da salvare saranno inseriti in un RedBlack tree, per poi essere successivamente salvati ordinatamente in un array pre allocato da ritornare.
- se `IDX_RMUL_SYMB_RBTREE` è configurata a `FALSE`, tutti gli indici da salvare saranno inseriti (in ordine casuale dipendente dai non zeri delle matrici considerate) direttamente nell'array da ritornare per poi essere ordinati.

Entrambi questi approcci hanno un costo computazionale pari a $O(nlogn)$ dove n è il numero di indici inseriti.

3.2.3.2 Array di flag

Alternativamente, se la macro di configurazione `SPVECT_IDX_BITWISE` è settata a FALSE, ho realizzato un approccio più semplice, basato su un array di N *char*, dove l'indice di colonna i da inserire corrisponde ad un valore non zero nel i-esimo elemento dell'array. La gestione della versione `OutIdxs_` della funzione target è analoga al caso precedente.

3.2.4 Sp3MM: Calcolo simbolico diretto del triplo prodotto

Dato che il problema orinario da risolvere è il triplo prodotto tra matrici sparse per applicazioni come amg4psblas [DDF21], ho deciso di cercare di supportare anche la moltiplicazione diretta tra 3 matrici sparse, con un'estensione della formulazione row-by-row.
Seguendo la notazione introdotta inizialmente in 2.1, l'approccio seguito è derivato da quello seguito da [JPD15].
Una volta computata la riga $(R \cdot AC_i)_{r*}$ con la versione `OutIdxs_` di `SpMM_Row_Symb_`, questa viene riutilizzata direttamente nel prodotto con la matrice P, analogamente al caso di SpMM, per ottenere la dimensione della riga relativa della matrice risultante $AC_{i+1_{r*}}$ ed eventualmente anche altre informazioni.
Le versioni `OutIdxs_` sono necessarie per ottenere gli indici di colonne della riga $(R \cdot AC_i)_{r*}$, indispensabili per reiterare la formulazione row-by-row sulla matrice P.
L'operazione appena descritta è eseguita in parallelo per ogni riga di AC_{i+1}.

3.2.5 Pro e contro UpperBound vs Prodotto Simbolico Accurato

L'analisi delle differenze prestazionali delle implementazioni di SpMM con fase Simbolica Accurata o UpperBound sarà analizzata in seguito nel capitolo 6.

Di seguto qualche considerazione di carattere generale che è possibile trarre dall'analisi di questi approcci differenti al problema di Sp[3]MM.

1. L'UpperBound gode sicuramente di un costo computazione nettamente inferiore rispetto a qualsiasi implementazione di Prodotto Simbolico accurato.

2. Tuttavia, questo vantaggio iniziale per soluzioni basate su UpperBound, è associato alla necessità di dover salvare tutti i risultati intermedi in una struttura temporanea, di cui è necessario effettuare una copia nella matrice risultante.

   (a) Viceversa il Pro dotto Simbolico accurato permette di poter salvare tutti i risultati intermedi, ottenuti durante la fase Numerica, direttamente nella matrice risultante.

3. Il Prodotto Simbolico accurato di $A \cdot B$, basato su una formul azione derivata da gustavson [Gus78], ad esempio row-by-ro, soffre di un ulteriore svantaggio dovuto a dover accedere le righe B in un ordine casuale, derivato dagli indici degli elementi non zero della riga corrispondente di A.
   Questo causa potenzialmente un cattivo uso della memoria, come analizzato da [JPD15] ed indirettamente da [ZG20].

   (a) Viceversa, il costo addizionale legato allo svantaggio dell'uso di UpperBound menzionato al punto 2, comporta la copia di risultati intermedi nella matrice risultante in maniera contigua, come osservato da [JPD15]

4. In generale, l'overhead di memoria causato dall'UpperBound, potrebbe essere tale da rendere impossibile sfruttare su GPU una fase simbolica basata su UpperBound.

CAPITOLO 4

IMPLEMENTAZIONI OPENMP: PRODOTTO NUMERICO

Prendendo spunto dalle caratteristiche principali degli algoritmi descritti precedentemente in 2, in questa sezione vengono descritte implementazioni parallele derivate da gustavson [Gus78] con formulazione row-by-row 2.2 per il prodotto numerico tra matrici sparse con OpenMP, utilizzando diverse metodologie. Estendendo l'approccio row-by-row, ho realizzato anche una versione per computare un triplo prodotto numerico direttamente.
In questo capitolo ho organizzato i contenuti per trattare prima dei componenti di supporto e configurazioni del prodotto numerico, tra cui la differente gestione della memoria in base a quale tipo di prodotto simbolico si sta usando 3.1.

4.1 Accumulatore denso per le moltiplicazioni scalari intermedie

Come precedentemente visto, una formulazione row-by-row di SpMM prevede di computare una riga di C mediante la somma di moltiplicazioni scalari intermedie. Analizzando gli ottimi risultati ottenuti da [MMAPD08] precedentemente in 2.5.3.1, ho deciso di sfruttare l'idea di utilizzare un accumulatore denso per mantenere e sommare le moltiplicazioni scalari intermedie.
Riguardo la determinazione degli indici di colonna degli elementi non zero calcolati mediante l'accumulatore denso, l'articolo menzionato [MMAPD08] non specifica l'approccio usato nel dettaglio, ma solo che è possibile mantenere efficientemente gli indici inserendoli in un array quando l'elemento relativo nell'accumulatore denso era zero.
Tuttavia, quest'approccio è potenzialmente soggetto a reinserimenti di stessi indici in scenari in cui: i valori non zero della riga in uso di A comportano un particolare sequenziamento nelle somme delle moltiplicazioni scalari che portano in una determinata entry dell'accumulatore denso a passare da zero a una serie di valori che passano

nuovamente per zero (numerico).
Dato che la entry è tornata a zero, sommandoci un qualsiasi altro valore non zero comporterà di dover reinserire il corrispondente indice.
Per quanto questo scenario è discretamente improbabile, nel gestirlo si richiede di dover eliminare gli indici duplicati.
Al fine di evitare questa necessità, che potenzialmente richiede una ulteriore passata su tutti gli indici inseriti, ho deciso di riutilizzare l'idea del set di flag usata per il prodotto simbolico in 3.2.3.
Ogni indice di colonna di elemento non zero considerato durante la moltiplicazione scalare richiesta dalla formulazione row-by-row, verrà inserito in maniera del tutto analoga al caso del prodotto simbolico, incrementando un contatore se l'indice non era gia nella struttura ausiliaria di ricerca.
Le implementazioni delle bitmap per gestire set di flag saranno descritte in 5.3.

```
1 typedef struct{
2     double*  v;                    //nnz dense accumulator   (sparsellly filled)
3     idx_t*   nnzIdx;               //v's nnz value's indexes   (contiguosily filled)
4     idx_t    vLen;
5     SPVECT_IDX_DENSE_MAP nnzIdxMap; //nnzIdx's indexed inserted flag set struct
6 } ACC_DENSE;
```

Nel frammento di codice precedente è riportata la struttura utilizzata come accumulatore denso per una riga della matrice risultante C.
In `v` saranno accumulati i valori non zero ed i relativi indici di colonna saranno copiati consecutivamente in `nnzIdx` e settati a `true` in `nnzIdxMap`.

4.1.1 Trasformazione dell'accumulatore denso in un vettore sparso

L'accumulatore denso calcolato precedentemente deve essere trasformato in un formato sparso per poter essere inserito nella matrice target.
Per fare questa operazione di *sparsificazione* efficientemente viene utilizzato il vettore degli indici di `nnzIdx`, come mezzo per accedere solamente alle locazioni dell'accumulatore denso `v` di interesse, come effettuato nel ciclo a riga 4 della seguente funzione.

```
1 static inline void _sparsifyUB(ACC_DENSE* accV,SPACC* accSparse,idx_t startColAcc){
2     idx_t nnz = accV->nnzIdxMap.len;
3     sort_idx_t(accV->nnzIdx,nnz); //sort nnz idx for ordered write
4     for (idx_t i=0,j;    i < nnz;   i++){
5         j =  accV->nnzIdx[i];
6         accSparse -> JA[i] = j + startColAcc;
7         accSparse -> AS[i] = accV->v[j];
8     }
9     accSparse -> len = nnz;
10 }
```

Nel frammento di codice precedente è possibile notare anche come sia gestita la possibilità di convertire un accumulatore denso relativo ad una partizione di riga della matrice risultante, gestendo un shifting degli indici di colonna non zero relativi mediante la variabile `startColAcc` a riga 6 (necessario per partizionamenti bidimensionali del lavoro come verrà approfondito in 4.3.2).
Ordinando `nnzIdx` prima della sua copia nella struttura di destinazione, serve a

produrre una matrice sparsa in output con indici degli elementi non zero ordinati. La necessità di tale operazione dipende dall'uso della matrice successivo al prodotto e nel caso CSR la distizione tra formato ordinato e non è riportata da [CH19].

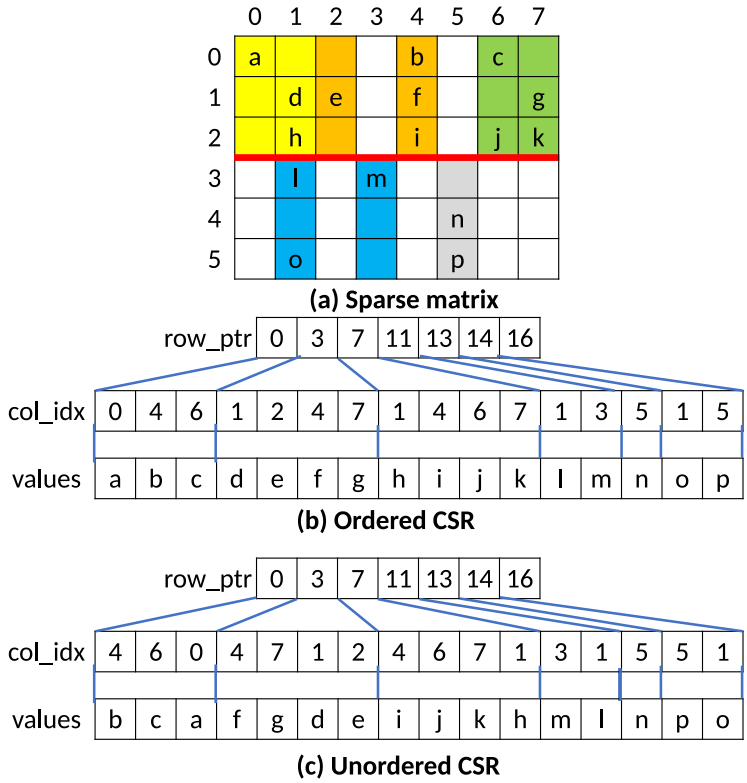


FIGURA 4.1: Distinzione tra matrici CSR con indici di colonna in `JA` (non) ordinati

Il costo dell'ordinamento degli indici della matrice risultante è ammortizzato su ogni partizione di riga assegnata ad ogni thread.
Ho deciso di implementare l'ordinamento mediante l'implementazione di quicksort presente nella `glibc` mediante la funzione `qsort`.

4.2 Gestione della memoria in base al tipo di Prodotto Simbolico usato

Come precedentemente detto in 2.3, è sempre preferibile utilizzare pre allocazioni in luogo di allocazioni dinamiche in implementazioni parallele.
In base al tipo di fase simbolica effettuata, possono esserci diverse possibilità per pre-allocare la memoria per l'operazione di SpMM e successivamente assegnarne

porzioni ai vari threads.

4.2.1 Possibilità offerte al Prodotto Numerico, in base al livello di output del Prodotto Simbolico

I livelli di output del prodotto simbolico definiti in 3, permettono di effettuare il prodotto numerico con vari livello di partizionamento del lavoro e di uso della memoria:

- un prodotto simbolico con output di livello 1 è sufficente a realizzare il prodotto numerico unicamente salvando i risultati intermedi in una struttura temporanea preallocata, di cui porzioni vengono assegnate ai thread dinamicamente. Dettagli di questo approccio saranno trattati a breve in 4.2.2.1
- un prodotto simbolico con output di livello 2 consente di effettuare il prodotto numerico con partizionamento monodimensionale del lavoro, dove le righe della matrice risultante possono essere salvate in una struttura temporanea preallocata o direttamente nella matrice risultante nel caso di prodotto simbolico accurato.
- un prodotto simbolico con output di livello 3 consente anche di effettuare il prodotto numerico con un partizionamento bidimensionale del lavoro, dove similmente al punto precedente è possibile scrivere le partizioni delle righe della matrice risultante in una struttura temporanea o direttamente nella matrice risultante se il prodotto simbolico è accurato.

4.2.2 Uso di una fase simbolica di tipo UpperBound

Nel caso di utilizzare una fase simbolica rapida basata su UpperBound in una implementazione parallela di SpMM , come già menzionato in 3.2.5, è necessario utilizzare durante la fase numerica una struttura temporanea per salvare i risultati dei prodotti.
Questa necessità è dovuta al fatto che ogni thread non ha informazione sulla esatta dimensione della porzione di matrice da calcolare e quindi non può scrivere i suoi risultati direttamente nell'output dell'operazione di SpMM.

La struttura intermedia che ho utilizzato è composta da una coppia di aree di memoria dimensionate con l'output della fase simbolica usata, destinate ai valori dei non zero e i relativi indici di colonna calcolati durante le moltiplicazioni scalari della fase numerica.
Nel seguito verranno indicate le aree per i non zero e i relativi indici del risultato di SpMM rispettivamente come *AS_tmp JA_tmp*.
Le partizioni di righe della matrice risultante calcolate verranno scritte da ogni

thread in aree contigue di questa struttura intermedia, salvandone gli indirizzi iniziali nella seguente struttura di supporto usata per l'operazione conclusiva di copia dei risultati intermedi nella matrice output (descritta in 4.2.2.3)

```
1//Sparse vector accumulator
2typedef struct{
3    double* AS;    //row part nnz values
4    idx_t*  JA;    //row part nnz indexes
5    idx_t   len;   //rowLen
6} SPACC;
```

Nella struttura precedente, con una notazione CSR, si rappresenta un generico vettore sparso con `len` elementi non zero in `AS` e relativi indici (di colonna nela caso CSR) in `JA`.
Il thread che avrà determinato una partizione di una riga della matrice target in due blocchi di memoria contigui nelle aree *AS_tmp JA_tmp*, ne annoterà i rispettivi indirizzi inziali nei puntatori `AS JA` e la lunghezza in `len`, rispettivamente a riga 3,4 e 5.
L'assegnazione delle aree di memoria della struttura intermedia per i risultati computati da ogni thread è gestita in modo dinamico o statico. È possibile selezionare quest'ultimo approccio a tempo di compilazione in luogo dell'altro definendo la macro `SPARSIFY_PRE_PARTITION` con valore TRUE.

4.2.2.1 Assegnamento dinamico di spazio per non zeri intermedi ai thread

Un approccio di assegnazione di porzioni di *AS_tmp JA_tmp*, simile ad effettuare allocazioni dinamiche, ma senza le criticità di performance citate in 2.3 è il seguente.
Ogni thread che necessita di *sparsificare* l'accumulatore denso determinato durante le moltiplicazioni scalari in un'area di memoria può incrementare **atomicamente** un indice relativo alla fine dell'ultima area di memoria assegnata ad un qualsiasi altro thread, catturandone il valore precedente.
Questa soluzione di gestione dello spazio intermedio ha il vantaggio di richiedere quasi nessuna informazione dal prodotto simbolico (è sufficiente un output di livello 1 secondo la numerazione descritta in 3) indipendentemente dal particolare tipo di partizionamento del lavoro che si sta utilizzando.
Segue la funzione per trasformare l'accumulatore denso in formato sparso con questo approccio.

```
1//row[Part] sparsified in a thread safe reserved area using atomics
2static inline void sparsifyUBNoPartsBounds
3  (SPMM_ACC* acc,ACC_DENSE* accV,SPACC* accSparse, ulong startColAcc){
4    idx_t nnz = accV->nnzIdxMap.len;
5    idx_t sparsifyStartV;
6    sparsifyStartV = __atomic_fetch_add(&(acc->lastAssigned),nnz,__ATOMIC_ACQ_REL);
7    accSparse -> AS = acc->AS + sparsifyStartV;
8    accSparse -> JA = acc->JA + sparsifyStartV;
9    _sparsifyUB(accV,accSparse,startColAcc);
10}
```

Nel frammento di codice precedente, è possibile vedere come a riga a 6 venga usata la builtin di gcc `__atomic_fetch_add` [SGDC21] per riservare al thread corrente una porzione di memoria di *AS_tmp JA_tmp* a riga 7 e 8, per scrivere i risultati intermedi

computati.
La primitiva `__atomic_fetch_add` permette di incrementare una locazione di memoria, ritornandone il valore non incrementato. Ulteriori dettagli su questo approccio e su alcune varianti saranno analizzati in 5.6

4.2.2.2 Assegnamento statico di spazio per non zeri intermedi ai thread

Un'alternativa semplice all'approccio precedente (4.2.2.1), è quello di effettuare una suddivisione per ogni (partizione di) riga della matrice risultante dello spazio allocato in *AS_tmp JA_tmp*, così da consentire ad ogni thread di poter scrivere i propri risultati intermedi in una spazio non soggetto a race-conditions.
Per realizzare questo è necessario avere conoscenza di un UpperBound per ogni (partizione di) riga della matrice risultante, e conseguentemente è necessario utilizzare un prodotto simbolico con un livello di dettaglio dell'output di livello 2 (3), seguendo la nomenclatura introdotta nel capitolo precedente 3.

4.2.2.3 Consolidamento risultati intermedi nel risultato finale

In conclusione all'operazione di SpMM con fase simbolica basata su UpperBound, è necessario raccogliere tutti i valori non zero e i relativi indici calcolati dalla struttura temporanea (*AS_tmp JA_tmp*) alla matrice risultante di output.
Per fare questo è sufficiente copiare solo gli elementi non zero scritti nella struttura temporanea mediante le strutture ausiliare `SPACC`, descritte in 4.2.2.
Segue la funzione per radunare tutti i risultati intermedi relativi a partizioni di righe della matrice risultante nell'output finale dell'operazione di SpMM.

```
1 inline int mergeRowsPartitions(SPACC* rowsParts,spmat* mat,CONFIG* conf){
2   //offsets for preparing copy
3     idx_t nzNum=0,j,rLen,idx,partsNum = mat->M * conf->gridCols;
4     idx_t* rowsPartsOffsets; //partsNum offsets for each row's partitiong of mat
5     for (idx_t r=0; r<mat->M; r++){ ///count nnz entries offsets per row's part.
6         for (j=0,rLen=0; j<conf->gridCols; j++){
7             idx = IDX2D(r,j,conf->gridCols);
8             rowsPartsOffsets[idx]=nzNum+rLen; //part start=prev accumulated end
9             rLen += rowsParts[idx].len;
10        }
11        nzNum += rLen;
12        mat->IRP[r+1] = nzNum;
13    #ifdef ROWLENS
14        mat->RL[r] = rLen;
15    #endif
16    }
17    mat->NZ = nzNum;
18  ...
19  //copy
20    idx_t pLen; //omp for aux vars
21    #pragma omp parallel for schedule(static) private(pLen)
22    for (idx_t i=0;  i<partsNum; i++){
23        pLen = rowsParts[i].len;
24        memcpy(mat->AS + rowsPartsOffsets[i],rowsParts[i].AS,pLen*sizeof(*(mat->AS)));
25        memcpy(mat->JA + rowsPartsOffsets[i],rowsParts[i].JA,pLen*sizeof(*(mat->JA)));
26    }
27    return EXIT_SUCCESS;
28 }
```

Come è possibile vedere nel frammento di codice precedente, l'operazione di copia dei risultati intermedi nella matrice risultante è divisa in 2 fasi.
Tra riga 5 e 16, viene cumulata la dimensione di tutti i blocchi di non zero computati

nel prodotto numerico in un vettore ausiliario `rowsPartsOffsets` e nella componente `IRP` della matrice risultante.
Successivamente la copia vera e propria avviene in parallelo a livello di singola partizione di riga risultante nel ciclo a riga 22 mediante delle `memcpy`.
Ad ogni iterazione si preleveranno i non zero scritti nella struttura temporanea realativi alla partizione $i \ mod$ `conf->gridCols`-esima della $\left\lfloor \frac{i}{\text{conf}\rightarrow\text{gridCols}} \right\rfloor$ riga dalla struttura temporanea, copiandoli nella posizione appropriata della matrice risultante.

4.2.3 Uso di una fase simbolica accurata

Nel caso di utilizzare un Prodotto Simbolico accurato, è possibile *sparsificare* gli accumulatori densi (descritti in 4.1) ottenuti sommando le moltiplicazioni scalari direttamente nella matrice risultante.
Per fare questo è necessaria una piccola operazione di inizializzazione della matrice di output andando a popolare il vettore `IRP`, con la cumulazione del numero di non zeri delle (partizioni di) righe determinato durante la fase simbolica.

4.3 Esecuzione del Prodotto Numerico

In questa sezione verranno analizzate metodologie per effettuare l'operazione di SpMM in parallelo, considerando vari approcci di suddivisione dei dati tra i thread. Come già detto l'uso di una fase simbolica accurata o basata su UpperBound, discrimina la necessità o meno di dover utilizzare una struttura intermedia durante il prodotto numerico. Oltre questo dettaglio non ci sono particolari differenze nelle due implementazioni della fase numerica e per questo si analizzeranno nel seguito le implementazioni che sono supportate da un prodotto simbolico basato su UpperBound.
La formulazione row-by-row, come indicato in 2.2, calcola la riga c_{i*} della matrice risultante sommando i vettori ottenuti dalle moltiplicazioni scalari tra: gli elementi $a_{ik} \in a_{i*}$ della corrispondente riga della matrice A e le righe $b_{k*} \in B$ relative agli indici di colonna k degli elementi a_{ik}. Questa operazione è esprimibile in maniera compatta come: $c_{i*} = \sum_{k \in I_i(A)} a_{ik} * b_{k*}$.
È possibile derivare il calcolo della partizione di colonne $c_z - c_w$ della riga c_{i*} limitando le moltiplicazioni scalari $a_{ik} * b_{k*}$ alle relative partizioni di colonne della matrice B. L'operazione appena descritta è esprimibile in maniera compatta seguendo la notazione introdotta 2.1 come:
$c_{i\ c_z - c_w} = \sum_{k \in I_i(A)} a_{ik} * b_{k\ c_z - c_w}$.

4.3.1 Partizionamento monodimensionale

Un semplice partizionamento delle matrici per parallelizzare SpMM, è quello di assegnare righe intere o blocchi di esse della matrice da calcolare *C* ai thread.
Nella terminologia introdotta precedentemente in 2.5, equivale ad assegnare gruppi di Layers del cubo di lavoro ai vari threads.
Segue una parte del codice per effettuare l'operazione di SpMM con un partizionamento dei dati di blocchi di righe della matrice A e C:

```
1 #pragma omp parallel for schedule(runtime) private(acc,startRow,block)
2 for (b=0;   b < cfg->gridRows; b++){
3     block    = UNIF_REMINDER_DISTRI(b,rowBlock,rowBlockRem);
4     startRow  = UNIF_REMINDER_DISTRI_STARTIDX(b,rowBlock,rowBlockRem);
5     acc      = accVects + omp_get_thread_num();
6     for (ulong r=startRow;  r<startRow+block;  r++){
7         for (ulong c=A->IRP[r]-OFF_F; c<A->IRP[r+1]-OFF_F; c++)
8             CAT(scSparseRowMul_,OFF_F)(A->AS[c], B, A->JA[c]-OFF_F, acc);
9         //trasform accumulated dense vector to a CSR row, in a tmp storage struct
10      #if SPARSIFY_PRE_PARTITIONING == T
11        _sparsifyUB(acc,outAccumul->accs+r,0);
12        #else
13        sparsifyUBNoPartsBounds(outAccumul,acc,outAccumul->accs + r,0);
14        #endif
15        _resetAccVect(acc);   //rezero for the next A row
16    }
17 }
18 ///merge sparse row computed before in output matrix
19 if (mergeRows(outAccumul->accs,AB))    goto _err;
```

Nel frammento di codice precedente è possibile vedere come ogni thread determini il proprio blocco di righe su cui operare a riga mediante la macro `UNIF_REMINDER_DISTRI_STARTIDX`, che consente di determinare blocchi di dimensione il più uniforme possibile effettuando una suddivisione del resto della divisione intera tra il numero di righe e `cfg->gridRows`, ovvero il grado di partizionamento monodimensionale della matrice. Maggiori dettagli e considerazioni a riguardo di quest'approccio di distribuzione saranno descritti in 5.9.
Nel ciclo a riga 6, il thread accumulerà le moltiplicazioni scalari nel suo accumulatore denso determinato a riga 5, per poi *sparsificarne* la riga risultante con una politica di assegnamento dinamica o statica dello spazio temporaneo pre allocato, rispettivamente a riga 11 o 13 (come affrontato rispettivamente in 4.2.2.1 e 4.2.2.2).
Dopo aver calcolato e *sparsificato*, una riga della matrice risultante, è necessario resettare l'accumulatore denso relativo usato (riga 15).
Al termine del calcolo di tutte le righe, è necessario raccoglierle dallo spazio temporaneo usato alla matrice risultante a riga 19 (come visto in 4.2.2.3).

4.3.2 Partizionamento bidimensionale

Un partizionamento 2D dell'operazione di SpMM è quello di assegnare ai threads blocchi bidimensionali della matrice C ottenuti dalla moltiplicazione di blocchi di righe della matrice A e blocchi di colonne della matrice B. Questa tecnica è stata rappresentata graficamente in 2.10.
Nella terminologia introdotta precedentemente in 2.5, equivale ad assegnare gruppi di fibers del cubo di lavoro ai vari threads.

Le moltiplicazioni scalari saranno limitate a partizioni di colonne di B, conseguentemente è possibile utilizzare un accumulatore denso dimensionato sulla più grande partizione di colonne di B da analizzare.
Ho deciso di seguire questo approccio per risparmiare allocazioni temporanee di memoria, al costo di dover gestire uno *shifting* degl'indici introdotti nell'accumulatore denso durante le operazioni di somma di moltiplicazioni scalari e *sparsificazione* degli accumulatori densi.
Al fine di effettuare un partizionamento della matrice CSR B per blocchi di colonne ho realizzato due approcci, uno basato su una struttura ausiliaria di offset per accedere le partizioni della matrice originale ed uno basato sul copiare la matrice in sotto matrici corrispondenti alle partizioni da accedere. Dettagli sulle tecniche di partizionamento per colonne di una matrice CSR saranno analizzate rispettivamente in 5.7.1 e 5.7.2.
Dal momento che il codice relativo all'uso di questi due approcci nel Prodotto Numerico è molto simile segue una descrizione del caso più semplice basato sulla copia delle partizioni di B in sotto matrici.

Segue l'implementazione del Prodotto Numerico con partizionamento bidimensionale del lavoro, sfruttando il partizionamento delle righe di A descritto in 4.3.1 e il partizionamento per colonne di B che verrà descritto in 5.7.2

```
1#pragma omp parallel for schedule(runtime) private(accV,accRowPart,...)
2for (tileID = 0; tileID < gridSize; tileID++){
3    ///get iteration's indexing variables
4    //tile index in the 2D grid of AB computation TODO OMP HOW TO PARALLELIZE 2 FOR
5    t_i = tileID/cfg->gridCols;  //i-th row block
6    t_j = tileID%cfg->gridCols;  //j-th col block
7    //get tile row-cols group FAIR sizes
8    rowBlock = UNIF_REMINDER_DISTRI(t_i,_rowBlock,_rowBlockRem);
9    startRow = UNIF_REMINDER_DISTRI_STARTIDX(t_i,_rowBlock,_rowBlockRem);
10   startCol = UNIF_REMINDER_DISTRI_STARTIDX(t_j,_colBlock,_colBlockRem);
11
12   colPart = colPartsB + t_j;
13   accV = accVectors + tileID;
14
15   for (ulong r=startRow;  r<startRow+rowBlock;  r++){   ///compute (A*B)[:t_i:][:t_j:]
16       //iterate over nz col index j inside current row r
17       //row-by-row restricted to colsubset of B to get AB[r][:colBlock:]
18       for (ulong j=A->IRP[r]-OFF_F,c,bRowStart,bRowLen; j<A->IRP[r+1]-OFF_F; j++){
19           //get start of B[A->JA[j]][:colBlock:]
20           c = A->JA[j]-OFF_F; // column of nnz entry in A[r][:] <-> target B row
21           bRowStart = colPart->IRP[c];
22           #ifdef ROWLENS
23           bRowLen   = colPart->RL[c];
24           #else
25           bRowLen   = colPart->IRP[c+1] - bRowStart;
26           #endif
27           CAT(scSparseVectMulPart_,OFF_F)(A->AS[j],colPart->AS+bRowStart,colPart->JA+bRowStart,
28               bRowLen,startCol,accV);
29       }
30
31       accRowPart = outAccumul->accs + IDX2D(r,t_j,cfg->gridCols);
32       #if SPARSIFY_PRE_PARTITIONING == T
33       _sparsifyUB(accV,accRowPart,startCol);
34       #else
35       sparsifyUBNoPartsBounds(outAccumul,accV,accRowPart,startCol);
36       #endif
37       _resetAccVect(accV);
38   }
39}
40if (mergeRowsPartitions(outAccumul->accs,AB,cfg))  goto _err;
```

Ogni thread è delegato al calcolo di un blocco 2D della matrice C, identificato dagl'indici `t_i` `t_j`, moltiplicando un blocco di righe della matrice A e un blocco di colonne della matrice B, identificati dalle variabili in righe 8-12.
Come nel caso monodimensionale, i blocchi assegnati ai thread sono di dimensione il più possibile uniforme.
Nel ciclo a riga 15, il thread corrente computerà la `t_j`-esima partizione della riga `r` di C, in un accumulatore denso che verrà *sparsificato* a riga 32-36, tenendo traccia dell'indice inziale della partizione di B in `startCol` per gestire lo shifting degli indici necessario ad usare un accumulatore denso dimensionato alla partizione di riga più grande di B.
Infine, a riga 40, tutte le partizioni di righe calcolate verranno unificate dalla struttura temporanea `outAccumul` alla matrice di output `AB`, in maniera analoga al caso precedente monodimensionale.

4.3.3 Sp3MM: Calcolo numerico diretto del triplo prodotto tra matrici

Dato che il problema orinario da risolvere è il triplo prodotto tra matrici sparse per applicazioni come amg4psblas [DDF21], ho deciso di di supportare anche la fase numerica della moltiplicazione diretta tra 3 matrici sparse, con un estenzione della formulazione row-by-row .

L'approccio che ho seguito è derivato dal lavoro di [JPD15] ed è basato sul riuso diretto delle righe computate con la formulazione row-by-row, per determinare le righe della matrice finale, senza l'ausilio di una matrice contente un'operazione di SpMM intermedia.
Nel seguito si seguirà la notazione introdota inizialmente riguardo il triplo prodotto tra matrici sparse (2.1).
Il primo passo per l'operazione di Sp3MM è calcolare la riga $(R \cdot AC_i)_{r*}$ sottoforma di accumulatore denso `accRAC`, sommando i risultati delle moltiplicazioni scalari relative alla formulazione row-by-row .
Successivamente, indicizzando `accRAC` esattamente come fatto nella *sparsificazione* di SpMM (4.1.1), è possibile ottenere la riga della matrice target $AC_{i+1r*} = (R \cdot AC_i \cdot P)_{r*}$ reiterando l'operazione di SpMM tra la riga ottenuta sottoforma di accumulatore denso e la matrice P.
Di seguito il blocco di codice principale per realizzare questa operazione.

```
1#pragma omp parallel for schedule(runtime) private(accRAC,accRACP,c)
2for (ulong r=0;  r<R->M; r++){    //row-by-row-by-row formulation
3    accRAC  = accVectorsR_AC  + omp_get_thread_num();
4    accRACP = accVectorsRAC_P + omp_get_thread_num();
5    //computing (tmp) R*AC r-th row
6    for (idx_t j=R->IRP[r]-OFF_F; j<R->IRP[r+1]-OFF_F; j++)
7        CAT(scSparseRowMul_,OFF_F)(R->AS[j], AC, R->JA[j]-OFF_F, accRAC);
8    //forward the computed tmp row for the final matrix's row
9    for (idx_t j=0; j<accRAC->nnzIdxMap.len; j++){
10       c = accRAC->nnzIdx[j];
11       CAT(scSparseRowMul_,OFF_F)(accRAC->v[c],P,c,accRACP);
12   }
```

Come è possibile vedere, la `r`-esima riga di $R \cdot AC_i$ ottenuta a linea 7 sottoforma di accumulatore denso, viene immediatamente riusata per determinare la `r`-esima riga di AC_{i+1} a linea 11, indicizzandola come fatto dalle funzioni di *sparsificazione* analizzate in precedenza, nel ciclo a riga 9.

CAPITOLO 5

COMPONENTI DI SUPPORTO A SP[3]MM

In questo capitolo verranno analizzati diversi componenti ausiliari sviluppati ed integrati per supportare le operazioni di SpMM e Sp3MM.

5.1 Derivazione automatica di molteplici implementazioni mediante PreProcessore C

Sfruttando una serie di direttive del pre processore C, sono riuscito ad ottenere per molte funzioni scritte nei sorgenti, varie versioni differenti ottenute a tempo di pre-processamento, accessibili in altre funzioni mediante una segnatura modificata. La necessità principale di realizzare questo approccio è stato quello di supportare, con un'alta efficienza, l'integrazione del codice C in un progetto fortran come [FB12], come verrà trattato a breve in 5.1.1.
I vantaggi di usare il pre processore per ottenere molteplici versioni di una funzione sono diverse:

- la possibilità di escludere completamente sezioni di codice non utili ad una versione
- la possibilità di aumentare di molto il riuso del codice scritto
- rispetto ad incapsulare in sotto funzioni le differenze delle varie versioni da ottenere rispetto ad una funzione base:
  - si consente al compilatore di effettuare ottimizzazioni su ogni versione ottenuta in seguito al pre processamento, non possibili a run-time, come ad esempio semplificare un offset pari a zero da applicare ad operazioni di indicizzazione

  - evitare istruzioni di branching addizionali nel codice, (come analizzato in 3.2.1.1) che potenzialmente permettono ulteriori ottimizzazioni a tempo di compilazione.

Ho pubblicato un semplice esempio di questo approccio implementativo in `https://github.com/andreadiiorio/C_Compile_Multi_Implementation_Automatically`.

Per ottenere diverse implementazioni da una funzione è necessario:

- racchiudere le differenze richieste da ogni versione in una funzione *base o generica* con una serie di direttive `#if ... #endif` ,
  così da consentire al preprocessore l'aggiunta del codice necessario a modificare la funzione base.
- Le diverse versioni della funzione *generica* devono essere esportate ad altre funzioni mediante una segnatura differente.
  Per questo è possibile aggiungere un suffisso al nome ed eventualmente argomenti addizionali alla funzione *base*, mediante la concatenazione di stringhe del pre-processore C.
  Un possibile approccio per supportare la concatenazione di stringhe e macro di configurazione in questo contesto è quello di usare la macro `CAT` del seguente blocco di codice.

```
#define _STR(s) #s
#define STR(s) _STR(s)

//Concatenate preprocessor tokens A and B WITHOUT   expanding macro definitions
#define _CAT(a,b)    a ## b
//Concatenate preprocessor tokens A and B           EXPANDING macro definitions
#define CAT(a,b)    _CAT(a,b)
```

- Le funzioni generiche devono essere racchiuse in un sorgente dedicato, ed incluse da un altro mediante `#include`, per un numero di volte pari al numero di versioni che è necessario ottenere, ridefininendo le macro ausiliare di configurazione delle implementazioni differenti.
- per avere esportate le dichiarazioni delle versioni differenti realizzate, è possibile reiterare l'approccio appena descrito agli header files.

5.1.1 Supporto integrazione in progetto Fortran

Al giorno d'oggi esistono vari approcci per integrare un applicazione C in una Fortran e viceversa, grazie agli standard del linguaggio Fortran 2003 e 2018, come descritto in [MM18].
Una differenza sostanziale di questi due linguaggi è l'uso di una differente indicizzazione, dove il C è a base 0 e il Fortran è a base 1.

È necessario tenere a mente questa differenza dato che in ogni formato di memorizzazione sparso di matrici sono presenti degli indici relativi alla posizione dei valori non zero nella matrice.
Per supportare l'integrazione di questo lavoro in [DDF21], è stato necessario supportare efficientemente un passaggio tra l'indicizzazione dell'applicazione fortran a quella C per eseguire il prodotto, eseguendo poi il vicerversa nel ritorno all'applicazione chiamante.
Al fine di supportare efficientemente il cambio di questi indici ho sfruttato l'approccio descritto per ottenere 2 versione per ogni funzione, una che accetti l'indicizzazione nativa del C e un'altra che accetti l'indicizzazione del Fortran, mediante l'aggiunta di un suffisso numerico ad ogni nome di funzione come nell'esempio successivo.

```
1 spmat* CAT(spmmRowByRow_SymbNum_,OFF_F)(spmat* A,spmat* B, CONFIG* cfg){...}
```

Dalla funzione precedente, definita nel file `Sp3MM_CSR_OMP_Symb_Generic.c` per eseguire il prodotto numerico con un partizionamento del lavoro 1D verranno ottenute due versioni distinte per supportare le due indicizzazioni in base al valore della macro `OFF_F` al momento dell'inclusione del sorgente.
La doppia indicizzazione è realizzata dall'uso della macro `OFF_F` all'interno della funzione come costante sottratta ad ogni indice preveniente dal Fortran prima di utilizzarlo per indicizzare un qualsiasi vettore, ed avrà valore `1` nella versione da usare in un'applicazione Fortran o `0` per una applicazione C.
Un vantaggio immediato di avere una versione dedicata all'indicizzazione C è quello di avere la certezza quasi assoluta che la correzione degli indici con il valore `0` in `OFF_F` verrà ottimizzata via dal compilatore.
Inoltre, per favorire una interoperabilità tra le varie implementazioni per le operazioni di SpMM e Sp3MM ho definito le seguenti interfacce per le operazioni:

```
1 typedef spmat* (*SPMM_INTERF )  (spmat*,spmat*,CONFIG*);
2 typedef spmat* (*SP3MM_INTERF)  (spmat*,spmat*,spmat*,CONFIG*,SPMM_INTERF);
```

Dove `spmat` è una struttura con campi per supportare varie rappresentazioni sparse come CSR o ELL, mentre `CONFIG` contiene la configurazione per le esecuzioni delle operazioni in oggetto, come ad esempio la griglia di parallelizzazione del lavoro o il numero di thread da utilizzare.

Per supportare il lato Fortran dell'integrazione è stato sfruttato il modulo `iso_c_binding` per definire le interfacce e strutture con controparte nel codice C, in un modulo di supporto.

5.1.2 Generazione efficiente di diverse versioni di una funzione base

Estendendo la soluzione appena riportata per generare una coppia di implementazioni per gestire una doppia indicizzazione, ho realizzato un supporto alla generazione

di otto versioni differenti di una funzione generica nel modulo relativo al prodotto simbolico.
Per realizzare le varie combinazioni di versioni richieste di ogni funzione per generare i vari livelli di dettaglio dell'output del prodotto simbolico descritti in 3.2.1.1, ho usato il seguente approccio basato su redefinizioni di alcune macro di supporto:

```
#define OFF_F 0
  ///generate basic versions
  #include "Sp3MM_CSR_OMP_SymbStep_Generic.c"
  ///generate outIdxs versions
  #define OUT_IDXS  OUT_IDXS_ON
    #include "Sp3MM_CSR_OMP_SymbStep_Generic.c"
  #undef  OUT_IDXS
  ///generate colParts versions
  #define COL_PARTS COL_PARTS_ON
    #include "Sp3MM_CSR_OMP_SymbStep_Generic.c"
    //generate outIdxs AND colParts versions
    #define OUT_IDXS  OUT_IDXS_ON
    #include "Sp3MM_CSR_OMP_SymbStep_Generic.c"

  #undef OUT_IDXS
  #undef COL_PARTS
#undef OFF_F
#define OFF_F 1
...
```

Nel frammento di codice precedente vengono generate dal preprocessore C otto diverse versioni della maggior parte delle funzioni definite nel sorgente `Sp3MM_CSR_OMP_SymbStep_Generic.c` combinando la ridefinizione delle macro di configurazione `OUT_IDXS COL_PARTS OFF_F`.

5.2 Linux Kernel 5.10.85 RedBlack Tree Userspace porting

L'implementazione dei RedBlack Tree usati nel prodotto simbolico è ottenuta effettuando un porting in userspace dei moduli relativi del kernel Linux 5.10.85.

5.2.1 RedBlack nel kernel Linux

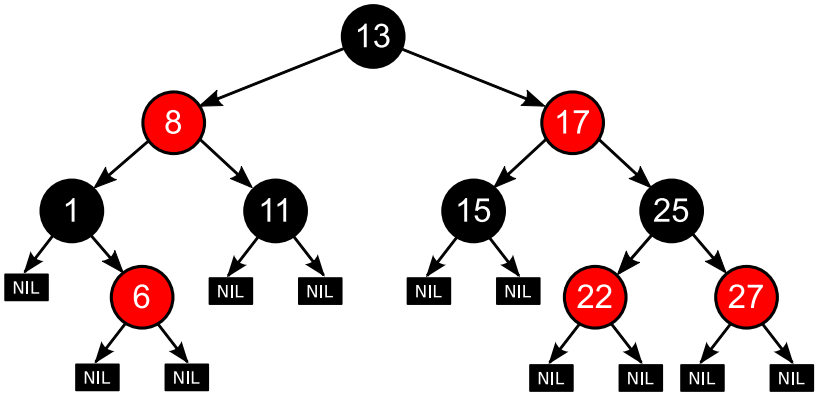


FIGURA 5.1: Rappresentazione grafica di un RedBlack Tree

I RedBlack Tree sono delle strutture di ricerca molto efficienti catalogabili come *Alberi binari auto bilanciati*.
La loro formulazione comprende diversi vincoli strutturali, da mantenre in seguito ad ogni operazione di inserimento o cancellazione, che garantiscono ottime proprietà di bilanciamento tra cui avere che:
la distanza tra la radice dell'albero e la foglia più lontana è minore o uguale della distanza tra la radice e la foglia più vicina.
Le proprietà di bilanciamento assieme ai vincoli strutturali garantiscono ai RedBlack Tree un costo computazione logaritmico al numero di elementi inseriti per ogni operazione.

Il kernel del sistema operativo Linux usa i RedBlack Tree per vari scopi, tra cui:

- le regioni di memoria virtuale (VMAs) associate ad un processo
- la gestione delle directory nel filesystem ext3
- la gestione dei timer ad alta risoluzione attivi
- la gestione dei pacchetti di rete nello scheduler `hierarchical token bucket`
- il tracciamento delle richieste negli schedulers `CFQ I/O` e `deadline`
- la gestione dei pacchetti nel driver per CD/DVD
- la gestione degli epoll file descriptors
- la gestione delle chiavi crittografiche

[Cor10; Cor06]

Un esempio dell'importanza dei RedBlack Tree è il primo punto della lista precedente, ovvero la gestione degli indirizzi associati ad un processo Linux.
Lo spazio degli indirizzi di memoria virtuale di un processo in Linux è implementato mediante una serie di intervalli di indirizzi denominati regioni di memoria, salvati sia in una lista collegata che in un RedBlack, contenuti nel `Memory Descriptor` di un processo.
Per supportare una ricerca efficiente della regione di memoria associata o vicina ad un indirizzo (e.g. `find_vma( )`) viene usato il RedBlack tree, mentre per operazioni di scansione completa dello spazio degli indirizzi viene utilizzata la lista collegata [BC05].

Segue una rappresentazione schematica delle strutture relative alla gestione delle regioni di memoria, con un focus sui RedBlack

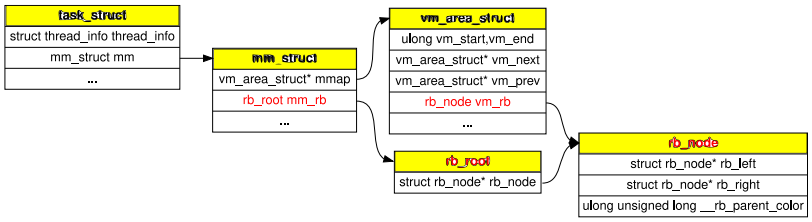


FIGURA 5.2: Rappresentazione delle strutture usate per la gestione dello spazio degli indirizzi di un processo evidenziando la presenza dei RedBlack Tree

---

Nella figura precedente è evidenziato come da un processo `p(task_struct)` è possibile accedere al suo `Memory Descriptor` associato (`mm_struct`), contente le varie regioni di memoria (`vm_area_struct`) relative ai vari intervalli di indirizzi collegati a `p`.
La gestione delle regioni di memoria è possibile sia mediante la strutturazione degli stessi in lista collegata (campi `vm_next vm_prev` in `vm_area_struct`), sia mediante RedBlack tree grazie ai campi evidenziati in rosso in 5.2

Dalle strutture `rb_root rb_node` dei RedBlack appena mostrate è possibile notare l'assenza campi contenenti chiavi o puntatori a strutture nodo. Questo fatto è dovuto alla scelta di ottimizzare l'implementazione di queste strutture eliminando layers di indirezione inutili all'esecuzione delle funzionalità principali di questi alberi. Da questa scelta degli sviluppati del kernel linux consegue la necessità di dover innestare le strutture `rb_node` all'interno di delle strutture nodo contenti un campo chiave adeguato ed dover implementare le funzioni di inserimento ed eliminazione nodi per ogni applicazione. La navigazione da una struttura `rb_node`

alla struttura nodo che la contiene è effettuabile semplicemente con la macro `container_of` .

Le tipologie principali di RedBlack tree presenti nel kernel linux sono:

1. normali identificati da una radice di tipo `rb_root`
2. left-cached identificati da una radice di tipo `rb_root_cached` caratterizzati dall'avere il nodo con chiave minima (più a sinistra nell'albero) salvato con un puntatore nella radice
3. augmented identificati da avere il suffisso `_augmented` nelle operazioni associate, caratterizzati dall'avere salvati in ogni nodo `n` dei dati derivati dal contenuto del sotto albero con radice in `n`.
   Con questo tipo di RedBlack Tree è possibile implementare un supporto efficiente alla gestione di range come chiavi, e.g. Interval Tree [Lan07]
4. latched identificati dalle strutture radice e nodo rispettivamente `latch_tree_root` `latch_tree_node` utili a supportare proprietà di consistenza in esecuzioni concorrenti lock-less, mediante la tecnica basata su multiple versioni dei dati *latch*

5.2.2 Porting in UserSpace

Ho effettuato il porting in UserSpace dei primi due tipi di RedBlack tree del kernel linux 5.10.27 menzionati precedentemente ed ho reso disponibile il lavoro ad altri usi nel mio repository `https://github.com/andreadiiorio/redblackTree_linux_userspace`.

Per effettuare il porting delle funzioni necessarie al prodotto simbolico con RedBlack Tree è stato necessario rimpiazzare varie dipendenze definite in altri moduli del kernel linux come le macro `container_of READ_ONCE WRITE_ONCE ...` o le funzioni di (de)allocazione con soluzioni alternative in userspace.
Sono state inoltre rimosse le versioni RCU delle operazioni sui RedBlack Tree per questioni di semplicità.

Per supportare la fase simbolica per l'operazione di SpMM è stata realizzata una seconda versione minimalizzata del porting, senza le funzioni non indispensabili al prodotto.

5.3 Uso Bitmaps per inserimento efficiente di indici

Per supportare l'inserimento efficiente di indici senza ripetizioni, in alcune versioni del prodotto simbolico descritte in 3.2.3.1 e nel gestire gl'indici di colonna negli accumulatori densi calcolati durante le moltiplicazioni scalari ho adottato una struttura di ricerca basata su un insieme di bitmaps.
La struttura contiene una zona di memoria in cui ogni bit può rappresentare la presenza di un indice specifico se è posto ad `1`.

L'approccio usato è in qualche modo correlato all'idea sfruttata in [Gmp], ovvero di supportare una variabile di dimensione arbitraria (in questo caso il numero massimo di indici inseribili) con molteplici variabili, *limb*, di una dimensione supportata dall'architettura target.

```
1 typedef unsigned __int128 uint128;
2 #if SPVECT_IDX_BITWISE == TRUE
3   #ifndef LIMB_T
4     #define LIMB_T uint128
5   #endif
6   typedef LIMB_T limb_t;
7   typedef limb_t* nnz_idxs_flags_t;
8   #define LIMB_SIZE_BIT ( sizeof(limb_t) * 8 )
9 #else //nnz idxs ar flags in a byte arry
10  typedef uchar* nnz_idxs_flags_t;
11 #endif
12 ...
13 typedef struct{                  //smart index keeping in a dense map
14   idx_t len;                  //num of nnz idx accumulated
15   /* nnz index presence packing, implict space enough for all possible indexes*/
16   nnz_idxs_flags_t idxsMap;
17   uint idxsMapN;              //either num of limbs or len of char flag array
18 } SPVECT_IDX_DENSE_MAP;
```

Nel frammento di codice precedente è riportata la struttura `SPVECT_IDX_DENSE_MAP`, che supporta il generico inserimento di indici in un area di memoria accessibile con `idxsMapN`, tenendone traccia del numero senza ripetizioni in `len`.
La macro di configurazione a riga 2 `SPVECT_IDX_BITWISE` determina se la struttura sarà realizzata mediante bitmaps o array di char.
Nel caso in oggetto di analisi, il tipo definito `nnz_idxs_flags_t` conterrà l'indirizzo di un array di bitmaps o *limb*.

La dimensione di ogni *limb* è definibile nel tipo `limb_t` a tempo di compilazione a riga 3, configurato di default all'estensione C di GCC per interi a 128bit con il tipo `__int128` [SGDC21].
Considerando un dimensionamento della struttura per contenere N indici, con *limb* di dimensione b bits saranno necessari $\lceil \frac{N}{b} \rceil$ *limb*. L'inserimento dell'indice i è effettuato mediante un'operazione di OR logico di un `1` nell' $\left\lceil \frac{i}{b} \right\rceil$-esimo *limb* nel $i \bmod b$-esimo bit.
Per supportare efficientemente il mantenimento del numero di indici inseriti, ad ogni inserimento viene associata una preventiva operazione di controllo se il bit target è già posto ad 1, in caso contrario avviene l'operazione OR e viene incrementato

un contatore.

```
1 static inline int spVect_idx_in(idx_t idx, SPVECT_IDX_DENSE_MAP* idxsMapAcc){
2   uint limbID   = idx / LIMB_SIZE_BIT; //idx's limb id
3   uint limbIdxID  = idx % LIMB_SIZE_BIT; //idx's pos in limb
4   limb_t idxPos   = ((limb_t) 1) << limbIdxID;
5   if (!( idxsMapAcc->idxsMap[limbID] & idxPos) ){
6     idxsMapAcc->idxsMap [limbID] |= idxPos;
7     idxsMapAcc->len++;
8     return 0;
9   }
10  return 1;
11 }
```

Nel frammento di codice precedente viene implementata l'operazione di inserimento di un indice nel set di bitmaps (riga 6) incrementando il contantore degli indici inseriti senza duplicati a riga 7 come descritto.

5.4 Configurazione chunksize dello scheduling dynamic OpenMP

Usare uno scheduling di tipo Dynamic in un ciclo parallelizzato con OpenMP può essere molto utile nel caso di problemi in cui il lavoro assegnabile ai thread ha una grande variabilità.
Nelle operazioni di SpMM e Sp3MM, la variabilità del lavoro è dovuta al pattern di sparsità dei non zero nelle matrici di input.

Lo scheduling Dynamic assegna un chunksize di dimensione pari ad 1 di default, il che può facilmente causare problemi di *false cache sharing*. Ovvero situazioni in cui il lavoro assegnato ad un thread `a` modifica una piccola area di memoria troppo vicina a quello di un altro thread `b`, al punto in cui le due aree mappano su linee di cache con una intersezione. In questo caso, una modifica del thread `a` sulla sua area di memoria invalida la copia della linea di cache intersezionata del thread `b`, causando un pessimo uso della memoria.

Lo scheduling Dynamic in confronto allo scheduling static di OpenMP, al costo di un overhead di istanziazzione e gestione maggiore, consente ad un thread che ha terminato il proprio lavoro di prenderne altro da una coda acceduta concorrentemente, evitando di restare bloccato in attesa sprecando risorse computazionali.
Conseguentemente, per il problema in analisi usare uno scheduling Dynamic con chunksize che sia un compromesso tra un valore piccolo e un chunksize statico, dato della divisione del numero di iterazioni per il numero di thread, può dare dei benefici prestazionali.
Per realizzare questa funzionalità ho realizzato una funzione che adatta dinamicamente il chunksize di un ciclo parallelizzato ad $\frac{\#\text{iterazioni}}{\#\text{thread} \cdot \texttt{FAIR\_CHUNKS\_FOLDING}}$ dove `FAIR_CHUNKS_FOLDING` è una macro configurabile a tempo di compilazione.

5.5 Configurazione automatica della griglia di partizionamento del lavoro

Nelle implementazioni parallele di SpMM descritte in 4 è prevista una suddivisione della computazione del risultato tra i vari thread mediante sia il chunksize configurato in openMP, sia tramite una griglia di partizionamento del lavoro.
Il chunksize dello scheduling openMP configurato determina quante iterazioni di un ciclo parallelizzato affidare ad un thread ed è configurato automaticamente in base al tipo di scheduling come descritto precedentemente in 5.4.
La griglia di partizionamento consiste in una suddivisione del risultato da calcolare in blocchi 1D o 2D.

Nel caso di implementazioni monodimensionali è possibile suddividere la computazione in blocchi di righe sia automaticamente mediante il chunksize di openMP, sia mediante un pre-partizionamento delle righe in blocchi all'interno del ciclo parallelizzato.
L'approccio usato in quest'ultimo caso è quello di suddividere le righe del risultato in blocchi di dimensione simile, in numero pari `SPMM_1DBLOCKS_THREAD_ITERATION_FACTOR` volte il numero di thread configurato per il calcolo.

Nel caso di implementazioni bidimensionali di SpMM è necessario strutturare il calcolo del risultato in blocchi 2D, corrispondenti a una suddivisione delle righe di A e delle colonne di B rispettivamente in `gridRowsXgridCols` blocchi.
Mediante le seguenti euristiche, la determinazione della griglia di partizionamento è stata resa configurabile in base al numero di thread configurato per l'esecuzione:

- automaticamente in una griglia di blocchi determinata mediante un porting che ho realizzato della funzione `MPI_Dims_create` da OpenMPI
- a partire da una suddivisione delle colonne di B da cui derivare un partizionamento delle righe di A pari a $\left\lceil \frac{numThread}{gridCols} \right\rceil$
- un partizionamento fisso, configurabile manulamente a tempo di compilazione con le macro `FIXED_2D_PARTITIONING_ROWS FIXED_2D_PARTITIONING_COLS`

5.5.1 porting funzione MPI_Dims_create da OpenMPI

Al fine di supportare una suddivisione del lavoro di SpMM per il numero di thread *nThread* configurato in una griglia di partizionamento 2D, il più quadrata possibile, ho realizzato un porting della funzione `MPI_Dims_create` da OpenMPI.

Nel contesto della programmazione a memoria distribuita, la funzione `MPI_Dims_create` consente di determinare una suddivisone bilanciata di un numero di processi in una topologia cartesiana n-dimensionale da utilizzare durante il calcolo [For21].
Nella determinazione della griglia di partizionamento per implementazioni bidimensionali di SpMM ho usato `MPI_Dims_create` per suddividere il numero di thread configurato nei parametri `gridRows gridCols`. Nel caso di utilizzare un numero di thread primo è impossibile ottenere una suddivisione esatta non monodimensionale, ma ho gestito questi casi suddividendo $nThread + 1$ in luogo di $nThread$, così da poter assegnare almeno un iterazione ad ogni thread.

5.6 Assegnamento dinamico di memoria ai thread *fence-less*

Nel caso di effettuare la *sparsificazione* di un accumulatore denso durante il prodotto numerico di SpMM, descritta precedentemente in 4.1.1, in assenza di informazioni sulla quantità di memoria necessaria ad ogni thread, è necessario assegnare dinamicamente partizioni di memoria (pre allocata), in maniera concorrente.

Un approccio semplice allo scopo è quello di utilizzare una qualche primitiva di sincronizzazione , funzionalmente simile ad un *lock*, per proteggere una sezione critica in cui si annota che il thread corrente si è riservato un segmento del blocco di memoria condiviso.
Tuttavia, quest'approccio potrebbe introdurre istruzioni di `fence` intorno alla sezione critica, causando una serializzazione deli accessi di memoria effettuati fin a quel momento.
Per ridurre al minimo l'overhead di sincronizzazione per realizzare questa assegnazione ho usato due approcci equivalenti.

5.6.1 Riservazione concorrente di memoria mediante built-in atomiche di GCC

Un approccio diretto al problema è quello di usare la built-in atomica offerta da gcc `type __atomic_fetch_add (type *ptr, type val, int memorder)` su una variabile atomica, contente l'indice dell'ultimo elemento assegnato del blocco di memoria condiviso ad un thread.

```
1 sparsifyStartV = __atomic_fetch_add(&(acc->lastAssigned),nnz,__ATOMIC_ACQ_REL);
```

Nel frammento di codice precedente, relativo alla sparsificazione di un accumulatore denso, viene mantenuto l'indice iniziale dello spazio di memoria condiviso non assegnato.

Ogni thread riserverà uno spazio di memoria mediante l'incremento atomico del numero di elementi da salvare (`nnz`) sulla variabile `lastAssigned`.
L'indirizzo iniziale del blocco di memoria riservato al thread sarà il valore precedente all'incremento di `lastAssigned` ed è salvato nella variabile `sparsifyStartV`.

L'uso di questa primitiva è da associare ad un modello di ordine di memoria `memorder`. Per l'operazione da realizzare dovrebbe essere sufficiente il modello `__ATOMIC_ACQ_REL`, che sostanzialmente crea una relazione di tipo *happens-before* tra le operazioni di *acquire* e *release* sulla variabile atomica [Ins12; SGDC21].

5.6.2 Riservazione concorrente di memoria mediante OpenMP

Un approccio del tutto equivalente è usare la clausola `capture` al construttuto `atomic` di openMP.
Segue una porzione di codice per realizzare la funzionalità in analisi.

```
#pragma omp atomic capture
{   //fetch and add like
    sparsifyStartV = acc->lastAssigned;
    acc->lastAssigned += nnz;
}
```

5.6.3 Confronto implementazione operazione atomica

Per confrontare le soluzioni, valutando anche altri modelli di memoria, ho effettuato un dissasemblamento del codice compilato con gcc 8.5.0.

```
sparsifyStartV = __atomic_fetch_add(&(acc->lastAssigned),nnz,__ATOMIC_SEQ_CST);
mov    -0x18(%rbp),%rax
add    $0x18,%rax
mov    -0x8(%rbp),%edx
lock   xadd %edx,(%rax)
mov    %edx,-0xc(%rbp)

sparsifyStartV = __atomic_fetch_add(&(acc->lastAssigned),nnz,__ATOMIC_ACQ_REL);
mov    -0x18(%rbp),%rax
add    $0x18,%rax
mov    -0x8(%rbp),%edx
lock   xadd %edx,(%rax)
mov    %edx,-0xc(%rbp)

#pragma omp atomic capture
{   //fetch and add like ....
  sparsifyStartV = acc->lastAssigned;
  acc->lastAssigned += nnz;
}
mov    -0x18(%rbp),%rax
add    $0x18,%rax
mov    -0x8(%rbp),%edx
lock   xadd %edx,(%rax)
mov    %edx,-0xc(%rbp)
```

Nel frammento di codice precedente vengono confrontati il dissasemblamento del codice configurato ad usare rispettivamente:

- la `__atomic_fetch_add` con modello di consistenza `__ATOMIC_SEQ_CST`, che dovrebbe offrire un ordinamento totale delle operazioni,
- la `__atomic_fetch_add` con modello di consistenza `__ATOMIC_ACQ_REL`

- il costrutto openMP `atomic` precedentemente visto.

È possibile notare come il codice Assembly prodotto sia sempre lo stesso, basato sull'uso di un'operazione di Exchange and Add con prefisso `LOCK`.
L'atomicità dell'implementazione è provata dalla presenza di questo prefisso, che come la documentazione intel riporta, consente al processore corrente di avere uso esclusivo di ogni memoria condivisa [Cor19].

5.7 Partizionamento bidimensionale di una matrice CSR

Per effettuare un partizionamento 2D di una matrice CSR è sufficiente dividere le colonne in gruppi ed accederne le righe. Dato che il vettore IRP della rappresentazione CSR permette di accederene facilmente le righe, avendo la conoscenza addizionale dei limiti di ogni partizione di colonne è possibile accedere blocchi bidimensionali della matrice.
Una rappresentazione grafica dell'operazione è raffigurata nell'immagine seguente.

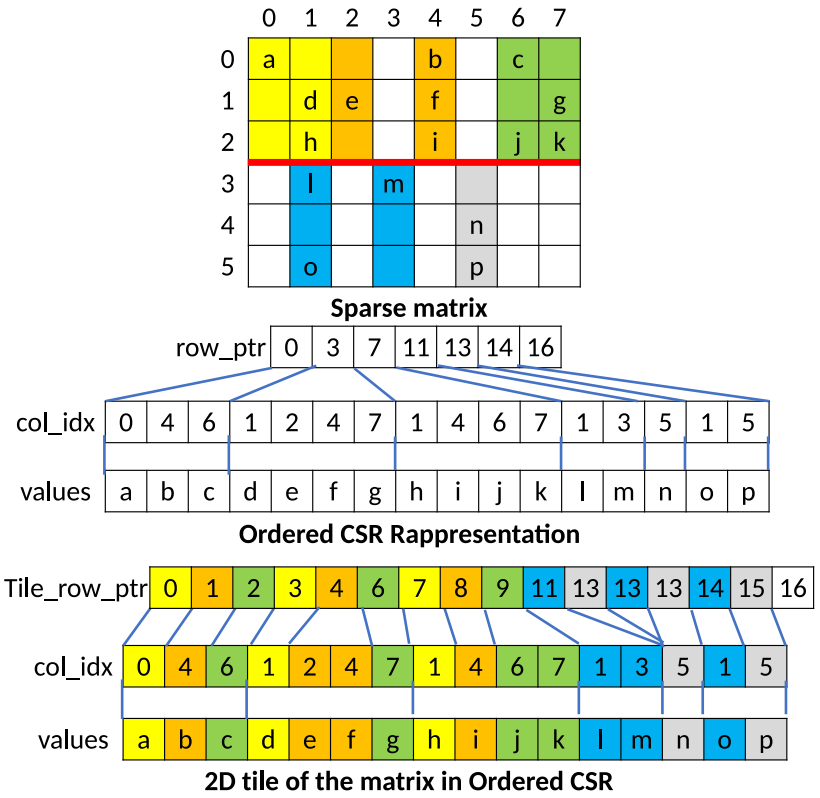


FIGURA 5.3: Rappresentazione grafica di un partizionamento 2D di una matrice CSR, [CH19]

La suddivisione delle colonne di una matrice CSR, necessaria per il suo partizionamento 2D, può essere effettuata *in loco* o mediante sotto matrici CSR separate con una struttura di offest di supporto.

Nel seguito saranno descritte due soluzione che ho realizzato per effettuare l'operazione in oggetto.

5.7.1 Partizionamento colonne in loco mediante offsets di supporto

Per accedere in loco una matrice CSR $M \times N$ in blocchi bidimensionali di $\frac{M}{gridRows} \times \frac{N}{gridCols}$, ho realizzato un supporto alla generazione di una matrice di offset $M \times gridCols$, dove l'elemento i, j è relativo all'inizio della j-esima partizione di colonne della i-esima riga.

```
1///OFFSETS COMPUTE FOR COL GROUPS -> O( A.NZ )
2for (ulong r=0, j=0;     r<A->M;     j=A->IRP[++r]-OFF_F){
3    offsets[ IDX2D(r,0,gridCols) ] = j;  //row's first gc start is costrained
4    for (ulong gc=1,gcStartCol;  gc<gridCols;  gc++){
5        gcStartCol = UNIF_REMINDER_DISTRI_STARTIDX(gc,_colBlock,_colBlockRem);
6        //goto GroupCols start entry,keeping A's nnz entries navigation (idx j)
```

```
7         while ( j < A->IRP[r+1]-OFF_F &&  A->JA[j]-OFF_F < gcStartCol )  j++;
8         offsets[ IDX2D(r,gc,gridCols) ] = j;  //row's gc group startIdx
9     }
10}
```

Nel frammento di codice precedente è possibile vedere come la matrice da partizionare è scansionata linearmente, salvandone l'indice di colonna corrente ogni volta che viene raggiunto o superato l'inizio di un gruppo di colonne.

5.7.2 Partizionamento colonne mediante sotto-matrici dedicate

Un approccio alternativo al precedente è quello di separare i gruppi di colonne della matrice da partizionare in sotto matrici separate. Successivamente è possibile accedere un blocco 2D della matrice originaria partizionando le righe (indirizzandole con IRP) di ogni sotto matrice ottenuta.
Ho realizzato due implementazioni di quest'approccio, uno basato sull'uso della struttura ausiliara vista nella sottosezione precedente, ed un'altra basata su una scansione diretta della matrice, simile al caso precedente.

```
1for (ulong r=0, j=0;     r<A->M;     j=A->IRP[++r]-OFF_F){
2     //navigate column groups inside current row
3     for (ulong gc=0,gcEndCol=0,i;  gc<gridCols ;  gc++,j+=i){
4         i = 0;  //@i=len current subpartition of row @r to copy
5         colPart = colParts + gc;
6         colPart->IRP[r] = colPartsLens[gc];
7         gcEndCol += UNIF_REMINDER_DISTRI(gc,_colBlock,_colBlockRem);
8         //goto next GroupCols,keeping A's nnz entries navigation ( index j+i )
9         while ( j+i < A->IRP[r+1]-OFF_F && A->JA[j+i]-OFF_F < gcEndCol ) i++;
10        memcpy(colPart->AS+colPart->IRP[r], A->AS+j, i*sizeof(*A->AS));
11        memcpy(colPart->JA+colPart->IRP[r], A->JA+j, i*sizeof(*A->JA));
12
13        colPartsLens[gc] += i;
14    #ifdef ROWLENS
15        colPart->RL[r] = i;
16    #endif
17    }
18}
```

Nel frammento di codice precedente è possibile vedere come scansionando la matrice da partizionare, si associ all'indice corrente l'inizio della partizione di colonne da riempirea(`j`) e la dimensione (`i`) della sotto riga attuale.Con queste informazioni è possibile effettuare una `memcpy` dei non zero relativi `gc`-esima partizione della `r`-esima riga della matrice (righe 10-11), dopo che l'indice di scanzionamento corrente della matrice `j+i` sia arrivato alla fine della sotto riga da copiare.

5.7.3 Confronto teorico delle due soluzioni

- L'approccio basato sulla generazione di sottomatrici per ogni partizione di colonna soffre di un overhead di inizializzazione superiore all'altro dato che oltre che scansionare la matrice per identificarne blocchi di colonne, è necessario effettuarne copie in altre zone di memoria.

- L'approccio basato sulla struttura di indicizzazione ausiliaria applicato alla realizzazione del prodotto numerico con partizionamento 2D del lavoro, soffre di una possibile penalità di memoria nell'accedere righe consecutive di una stessa partizione, dato che i non zero relativi non sono contigui in memoria.

5.8 Verfica correttezza delle implementazioni

Tutte le implementazioni parallele realizzate per Sp3MM sono state verificate mediante un confronto con una matrice risultante ottenuta esternamente o da una implementazione seriale di riferimento. In entrambi i casi, il confronto è considerato positivo se c'è una uguaglianza esatta tra le dimensioni delle matrici e gli indici degli elementi non zero (il componente `JA` del formato CSR) e se i valori floating point delle matrici sono uguali a meno di una soglia di tolleranza (configarata con $7\ 10^{-4}$.
È importante sottolineare la necessità di effettuare il confronto tra valori non zero delle matrici con una soglia, dato che le operazioni tra rappresentazioni floating-point sono soggette ad errori di approssimazione che possono portare a risultati diversi in base al sequenziamento delle operazioni effettuate (non prevedibile in una implementazione parallela) [Gol91]

5.8.1 implementazione seriale di riferiemento

L'implementazione seriale di riferimento è stata realizzata con una coppia di operazioni di SpMM realizzate con un approccio row-by-row , in modo molto simile alle implementazioni monodimensionali descritte in 4.3.1. A differenza della controparte parallela, l'implementazione seriale gode della grande facilitazione di poter scrivere le righe della matrice risultante direttamente nell'output dell'operazione senza una fase simbolica precedente, data la possibilità di accumulare le lunghezze delle righe calcolate precedentemente alla corrente.

Per validare gli output dell'implementazione seriale per alcuni input di piccola dimensione, è stato effettuato un confronto con una implementazione di riferimento per matrici dense: `CBLAS` in `http://www.netlib.org/blas/`, in seguito ad una trasformazione delle matrici da un formato sparso ad uno denso.

5.9 Distribuzione uniforme del lavoro tra i thread

Distribuire uniformemente il carico di lavoro tra i thread è un'operazione importante, che consente di minimizzare il divario tra i tempi di completamento dei vari thread e conseguentemente anche il tempo di esecuzione parallelo (dal momento che è

definito con l'istante di terminazione dell'ultimo thread).

Algoritmi paralleli basati su strutture sparse sono spesso accumunati da un'impossibilità di determinare efficientemente il carico di lavoro di effettivo relativo ad una partizione dell'input. Nonostante ciò, può essere utile distribuire il più uniformente possibile ogni partizione di dati e iterazioni tra i thread .
Considerando un input di dimensione n da suddividere tra t threads, avendo $div = \lfloor \frac{n}{t} \rfloor$ $rem = n \ mod \ t$, l'approccio seguito per minimizzare il divario delle partizioni assegnate è realizzato dalla seguente macro:
`#define UNIF_REMINDER_DISTRI(i,div,rem) ((div)+((i)<(rem)?1:0 ))` dove viene redistribuito il resto della divisione *rem* tra i thread.

CAPITOLO 6

ANALISI DELLE PERFORMANCE DELLE IMPLEMENTAZIONI REALIZZATE

Ho effettuato un analisi delle performance ottenute nelle varie implementazioni realizzate per Sp3MM con un insieme eterogeno di matrici di input.
Le prestazioni confrontate sono ottenute da valori mediati su 40 ripetizioni dalle varie implementazioni e configurazioni realizzate per Sp3MM descritte precedentemente.
I dati sono stati raccolti mediante un programma di test (`Sp3MM_test.c`) automatizzando l'esecuzione di tutte le implementazioni realizzate dato un set di matrici di input.
I log del programma di test sono stati parsati ed analizzati mediante la libreria di analisi Pandas [MPDT22].
Tutti i test sono stati effettuati sul server dell'Ateneo, caratterizato dai seguenti parametri:

- OS: CentOS Stream 8
- CPU: Intel Xeon Silver 4210 2.20 GHz 40 core
- RAM: 64GB

6.1 Matrici di input utilizzate

Il problema originario risolto è un sistema lineare ottenuto da una discretizzazione alle differenze finite di una PDE di secondo ordine, utilizzando una griglia regolare e le condizioni al contorno di Dirichlet.
Le matrici sparse utilizzate nei test sono relative ai tripli prodotti di Galerkin in due configurazioni di generazione degli aggregati in AMG4PSBLAS: *VanekBrezina* e *Matching*, entrambe con e senza un operazione di smoothing associata.
Le caratteristiche del problema di partenza da cui sono derivate i gruppi di matrici, comportano un pattern di sparsità degli elementi non zero di tipo diagonale come è visibile nelle figure seguenti.

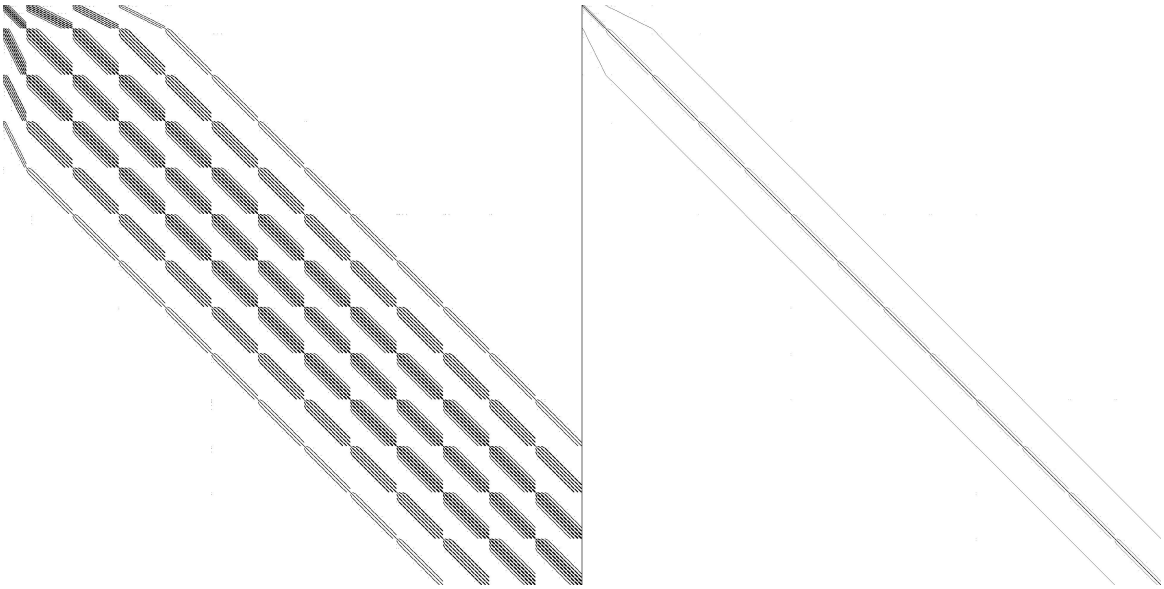


FIGURA 6.1: rappresentazione degli elementi non zero di matrici ottenute con il metodo Matching, con opzione smoothed a sinistra, unsmoothed a destra

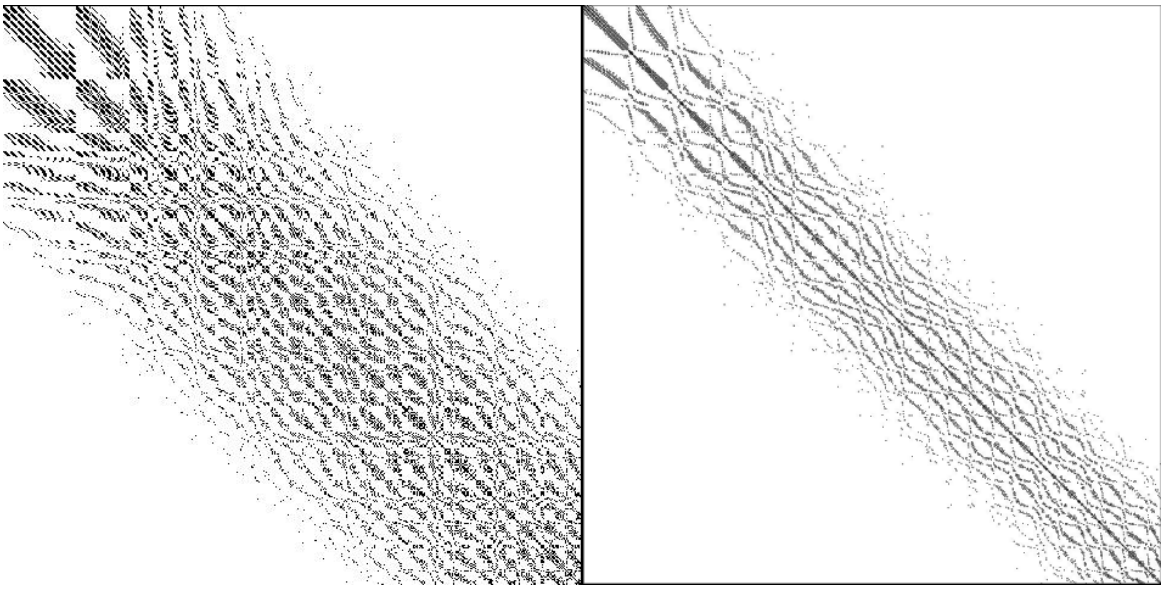


FIGURA 6.2: rappresentazione degli elementi non zero di matrici ottenute con il metodo VanekBrezina, con opzione smoothed a sinistra, unsmoothed a destra

Data l'eterogeneità degl'input utilizzati, ho raggruppato le matrici di test in quattro gruppi in base alla tecnica di generazione degli aggregati e l'uso dello dello

smoothing.
I gruppi di input utilizzati sono quindi:

1. VanekBrezina,Smoothed
2. VanekBrezina,Unsmoothed
3. Matching,Smoothed
4. Matching,Unsmoothed

6.2 Configurazioni considerate

Tra i svariati parametri configurabili a compile-time o a run-time, è stata effettuata un analisi di tutte le implementazioni realizzate variando:

1. il tipo di assegnamento ai thread dello spazio intermedio in implementazioni basate su UpperBound
2. la dimensione delle bitmap utilizzate in alcune implementazioni del prodotto simbolico
3. il numero di thread da utilizzare
4. il tipo di scheduling openMP: static o dynamic adattato (descritto in 5.4)

Inoltre, sono state determinate le performance:

1. di ogni implementazione, con configurazione ottimale, per ogni categoria di input (descritte in 6.1)
2. della migliore implementazione per ogni input, confrontadole con le performance ottenute con un implementazione seriale di riferimento

Altri parametri, come la griglia di partizionamento utilizzata, sono determinati automaticamente in base al numero di thread configurato, come descritto in 5.5.
In particolare per implementazioni bidimensionali, la griglia di partizionamento impiegata è stata determinata con l'approccio basato sulla funzione `MPI_Dims_create`, descritto in 5.5.1.

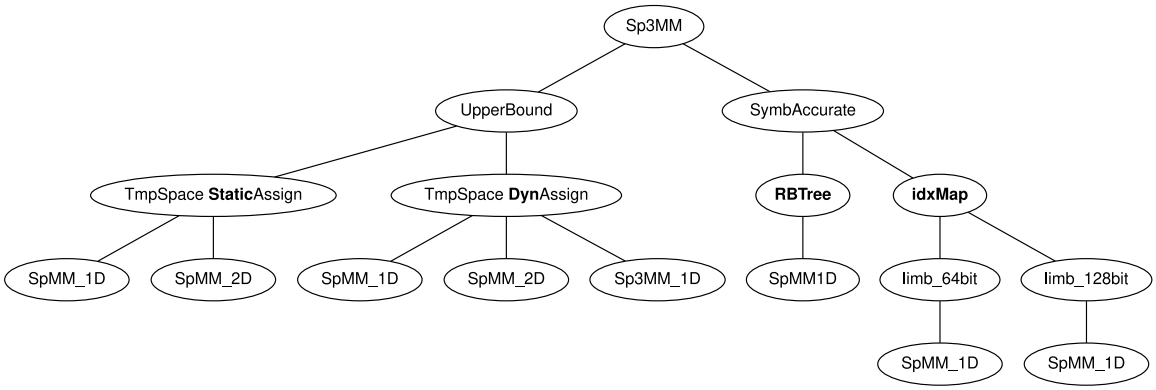


FIGURA 6.3: Diagramma riassuntivo delle varie configurazioni delle implementazione realizzate considerate nei test con scheduling openMP static e dynamic adattato

Tutti i test nel seguito ad eccezione di quelli menzionati nella sezione 6.6, sono stati effettuati con il numero massimo di thread possibili, ovvero 40 sull'ambiente di esecuzione utilizzato.

6.3 Determinazione della migliore configurazione

Nel seguito viene determinata la migliore configurazione, mediando le performance misurate nelle implementazioni relative alle configurazioni considerate, su tutti gli input disponibili e tutti i tipi di scheduling openMP

6.3.1 Migliore assegnamento di spazio intermedio per implementazioni UpperBound

Come analizzato in 3.2.5, implementazioni parallele di SpMM basate su una fase simbolica di tipo UpperBound, necessitano inevitabilmente di uno spazio temporaneo per i risultati intermedi, che ho pre-allocato al blocco di esecuzione parallelo ed assegnato ai thread in modo statico o dinamico, come descritto in 4.2.2.2 ed approfondito in 5.6.

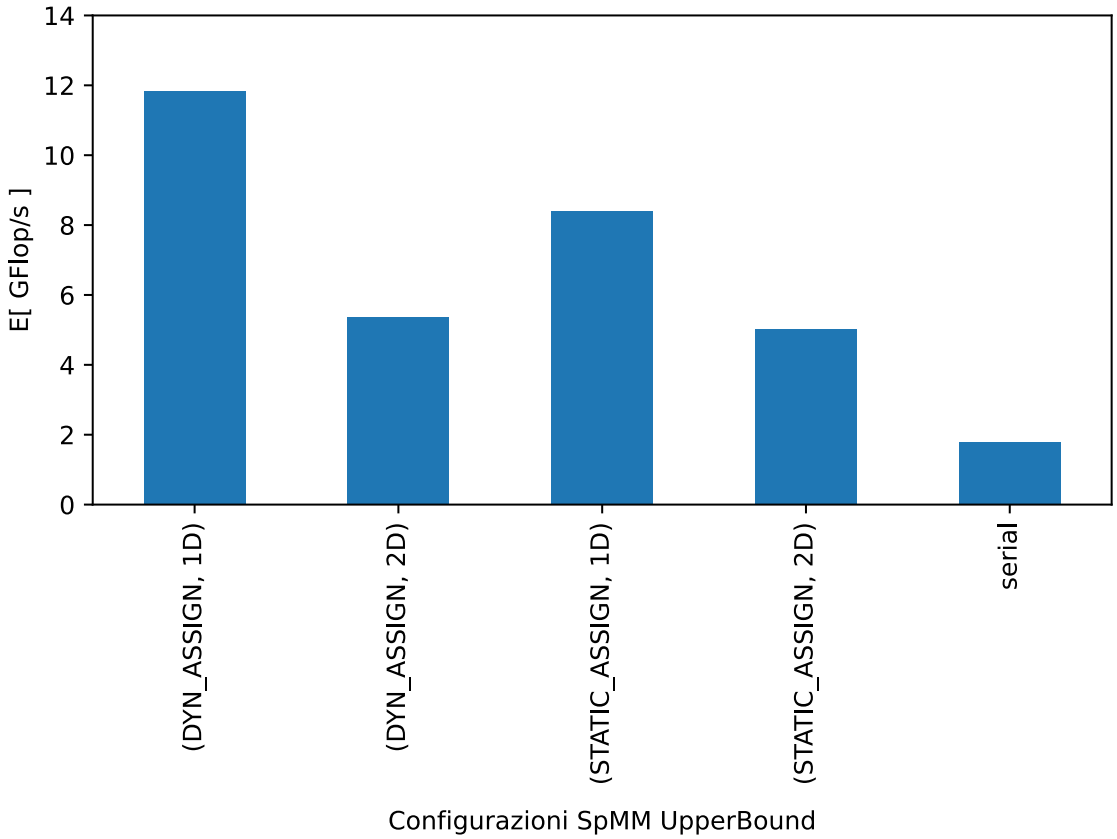


FIGURA 6.4: Confronto tra le perfomance medie delle implementazioni UpperBound di SpMM, su tutti gli input considerati, in base a tipo di assegnamento dello spazio intermedio ai thread (statico o dinamico) e al tipo di implementazione usata (con partizionamento del lavoro 1D o 2D)

Come è possibile vedere nella figura 6.4, un assegnamento dinamico dello spazio intermedio ai thread da dei vantaggi per implementazioni monodimensionali, mentre ha performance pressochè simili nel caso di implementazioni bidimensionali.

6.3.2 Migliore dimensione delle bitmap per implementazioni con prodotto simbolico accurato

La versione del prodotto simbolico accurato basata su bitmap di indici, descritta in 3.2.3 ed approfondita in 5.3, prevede l'uso di un vettore di variabili di tipo `limb_t` di dimensione configurabile a tempo di compilazione con la macro `LIMB_T`.

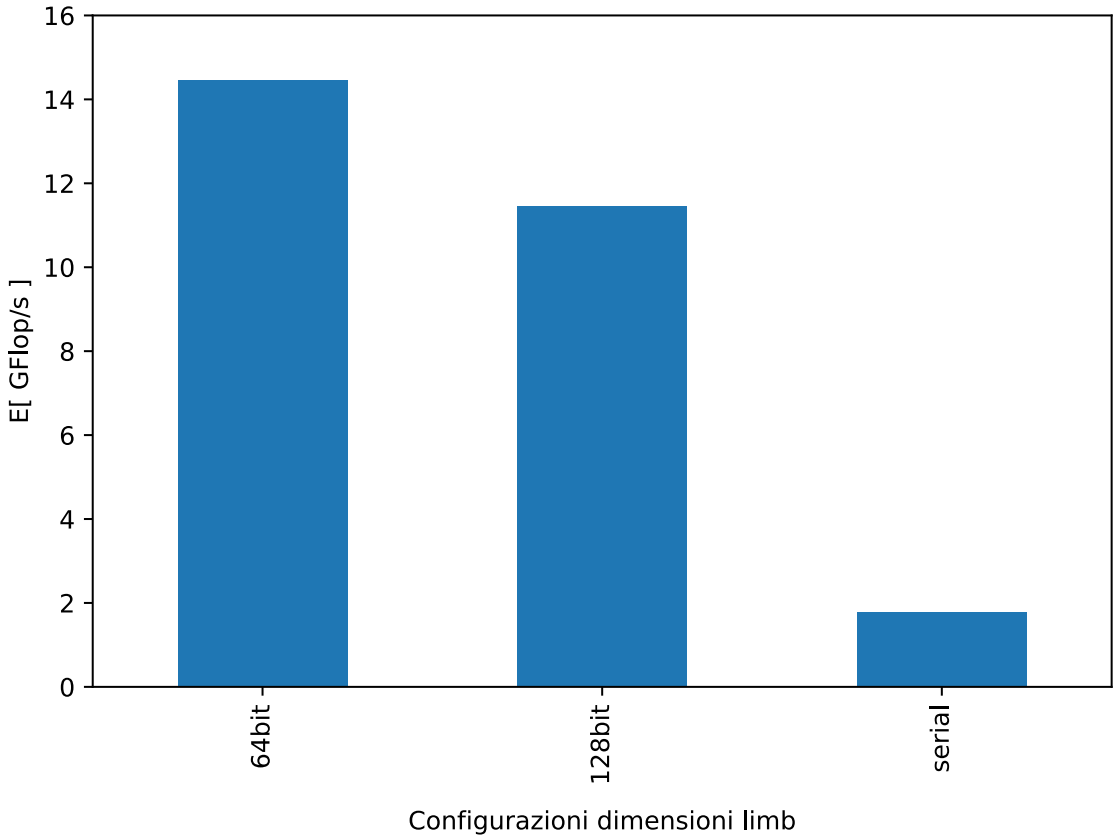


FIGURA 6.5: Confronto tra le perfomance medie di Sp3MM con fase simbolica accurata mediante bitmaps di indici, su tutti gli input considerati, variando la dimensione delle singole bitmap tra 64 e 128 bit

Dalla figura 6.5 è possibile notare come usare bitmap di 64 bit può può fornire un vantaggio prestazionale rispetto all'uso di bitmap a 128 bit.

6.4 Migliore implementazione per classe di input

Considerando la miglior configurazione determinata in 6.3, ho effettuato un confronto delle prestazioni medie delle varie implementazioni di Sp3MM per ogni classe di input disponibile (descritte precedentemnte in 6.1)

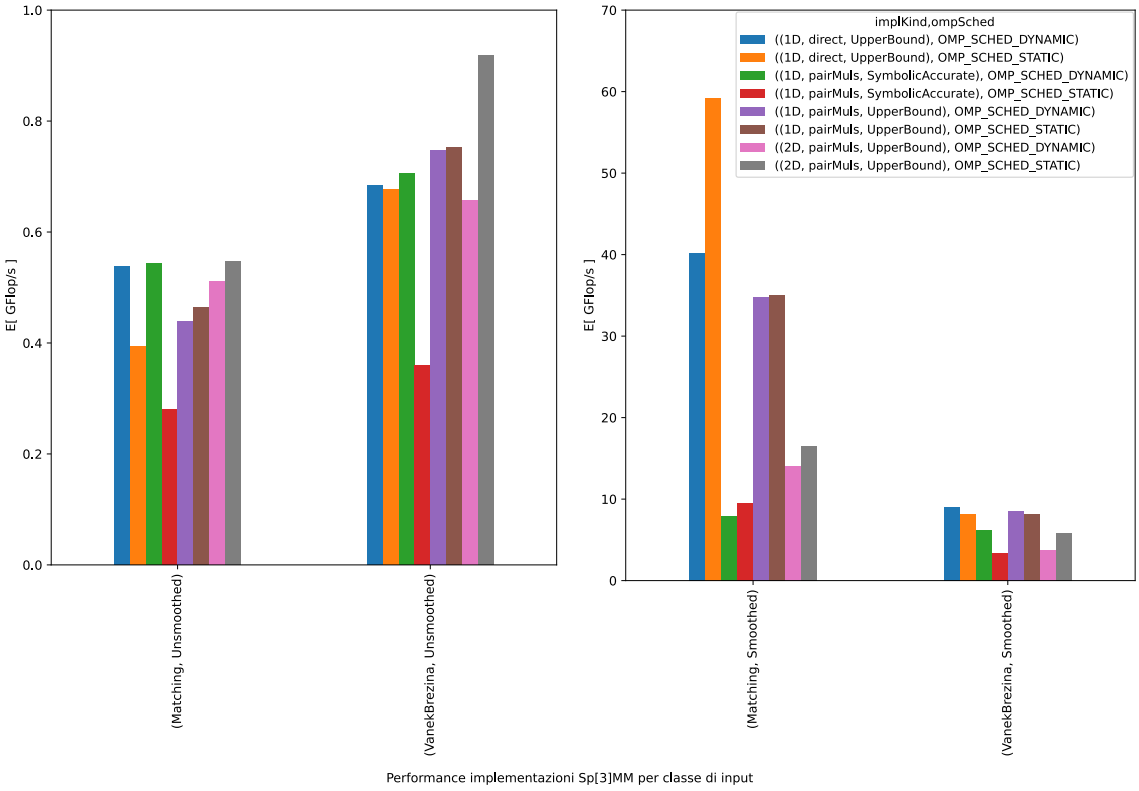


FIGURA 6.6: Confronto tra le perfomance medie della miglior configurazione per le implementazioni di Sp3MM per ogni classe di input

Come è possibile notare dalla figura 6.6, si hanno performance molto diverse in base alla classe di input considerata. In particolare, le implementazioni su matrici di tipo *Unsmoothed* soffrono di prestazione nettamente inferiori di quelle su matrici *Smoothed*.
Le implemenetazioni con performance migliori variano tra le classi di input considerate, tuttavia è possibile osservare che:

- il triplo prodotto diretto dia prestazioni migliori degli altri in tutte le classi salvo *VanekBrezina,Unsmoothed*.
- la configurazione di scheduling openMP static sembra dare prestazioni migliori per le implementazioni bidimensionali, viceversa la controparte dynamic con chunk adattati da risultati migliori per implementazioni monodimensionali e per il triplo prodotto diretto.
- le implementazioni monodimensionali UpperBound performano meglio di quelle bidimensionali nelle classi di input *Smoothed*, viceversa per le classi *Unsmoothed*

6.5 Guadagno di performance rispetto ad una implementazione seriale

Come precedentemente analizzato le prestazioni di Sp3MM sono molto variabili in base al tipo di input, l'implementazione utilizzata e la configurazione applicata. Per questo motivo ho deciso di misurare il vantaggio prestazionale ottenuto dalla migliore implementazione parallela rispetto ad una implementazione seriale di riferimento (descritta in 5.8.1).

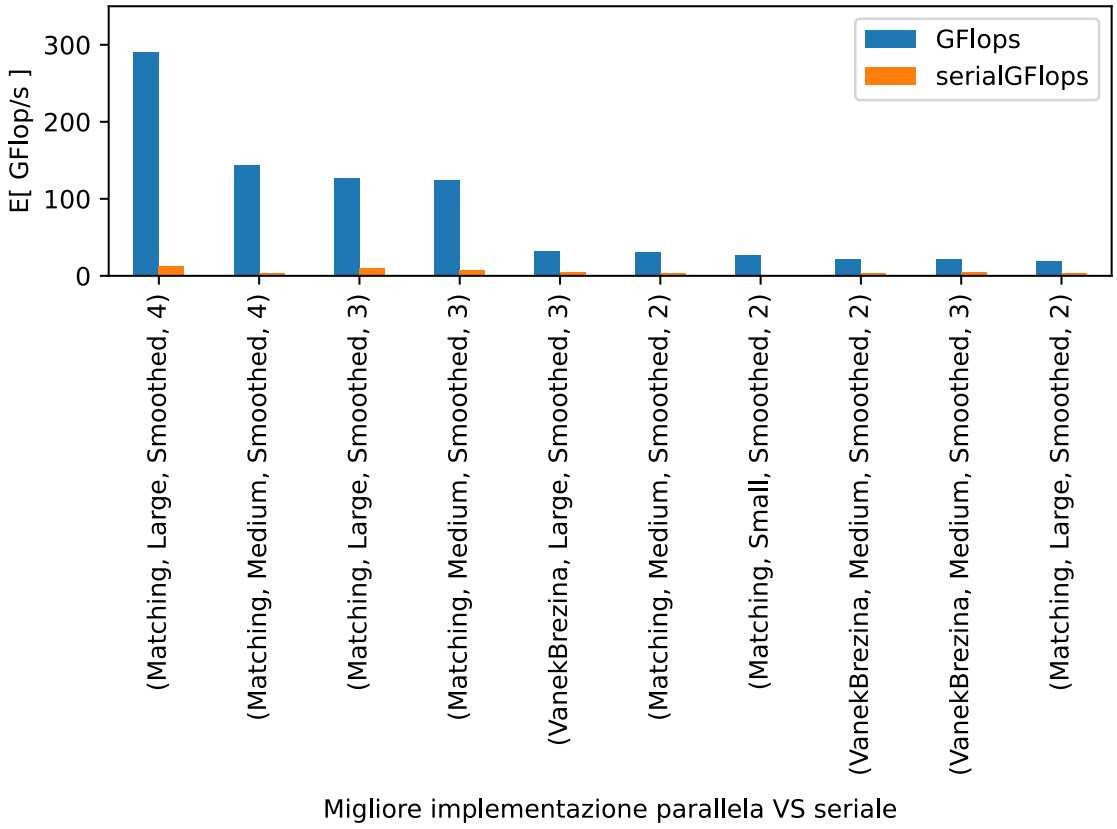


FIGURA 6.7:

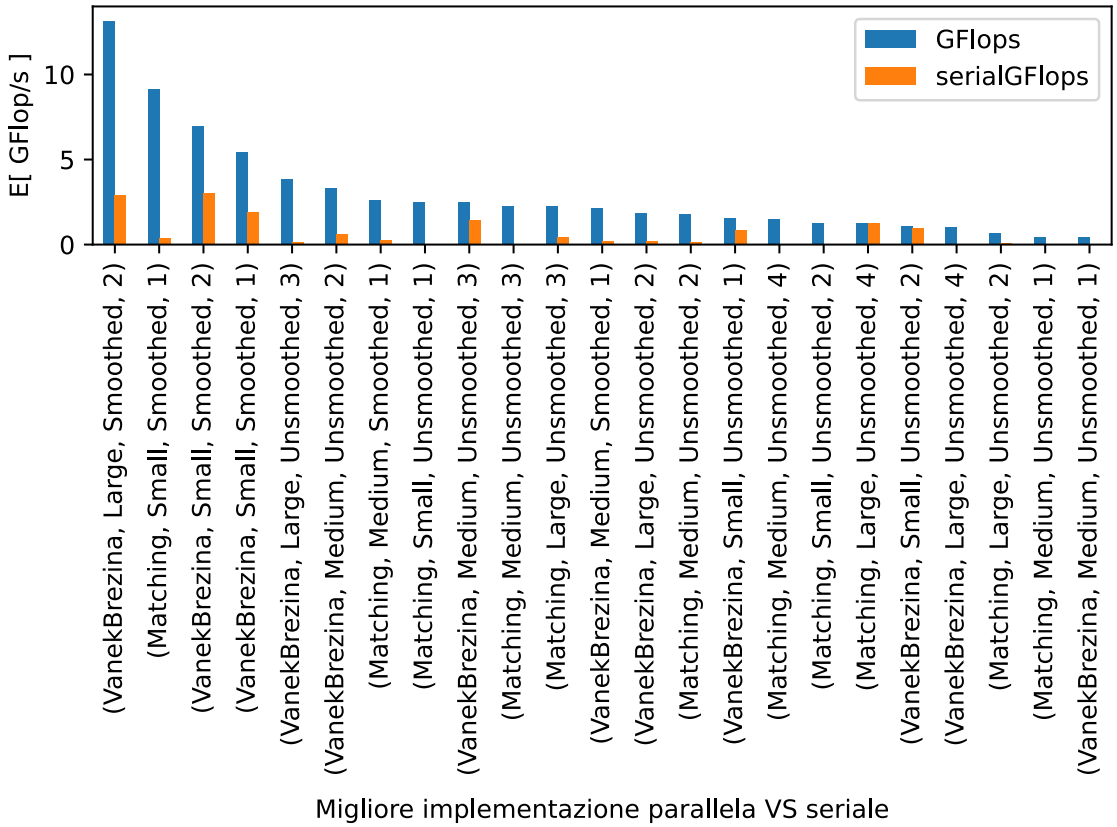


FIGURA 6.8:

Come è possibile notare dai grafici 6.7, 6.8, le implementazioni parallele di Sp3MM performano meglio dell'implementazione seriale per la stragrande maggioranza dei casi, in misura variabile in base all'input considerato.

6.6 Performance variando il numero di thread

La dimensione delle matrici di input e il loro pattern di sparsità, potrebbe comportare che le performance migliori per Sp3MM sono ottenute con un numero di thread diverso da quello massimo possibile.
Per questo motivo ho effettuato un analisi delle prestazioni misurate per alcune esecuzioni di Sp3MM al variare del numero di thread.

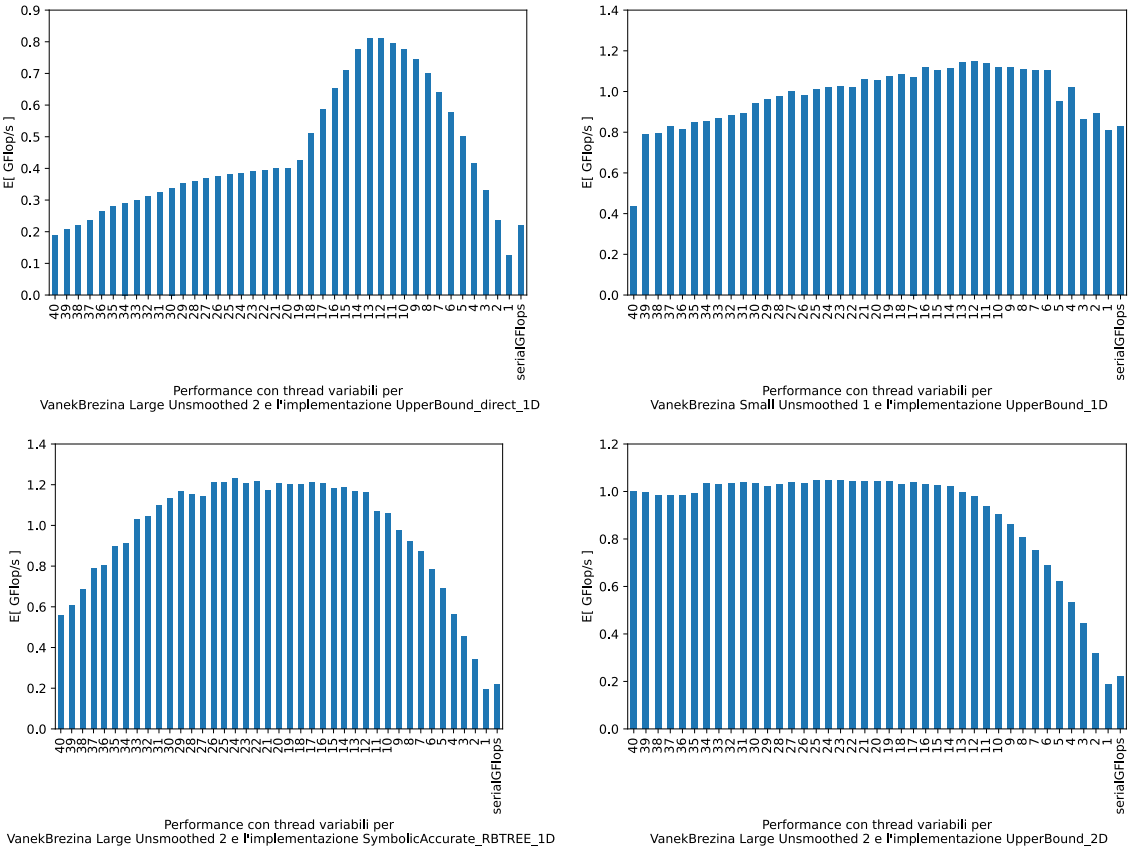


FIGURA 6.9: performance misurate su alcune esecuzioni di Sp3MM, variando il numero di thread impiegati

Come è possibile notare dai grafici 6.9, è possibile avere un guadagno di performance abbassando il grado di parallelismo, in misura differente in base all'implementazione e alla matrice considerata.

6.7 Conclusioni

Di seguito alcune considerazioni che è possibile trarre dalle analisi delle performance effettuate.
In generale, il pattern di sparsità dei valori non zero delle matrici di input può causare uno sbilanciamento nelle porzioni di lavoro assegnate ai thread in misura differente in base:

- all'approccio di divisione del lavoro usato,
- alla tipologia di implementazione usata (triplo prodotto diretto o uso di una fase simolica accurata)

- al tipo di scheduling openMP usato così come altre configurazioni

Uno sbilanciamento del carico di lavoro comporta un cattivo uso delle risorse computazionali, causando un peggioramento delle performance come come è possibile notare nell'analisi effettuate in 6.4.

Il livello di parallelismo usato può influire molto sulle performance, come visibile nell'analisi effettuata in 6.6, sia per le ragioni appena citate che per motivi legati all'overhead di schedulare un alto numero di thread per input piccoli.

L'overhead complessivamente introdotto da openMP può in generale comportare un costo superiore al beneficio ottenuto dall'esecuzione parallela per alcuni input, come visibile in pochi casi in 6.8. Inoltre, in 6.9 è possibile notare uno svantaggio prestazionale in alcune esecuzioni parallele con un solo thread rispetto all'esecuzione dell'implementazione seriale di riferimento, probabilmente dovuto all'overhead di inizializzazione e scheduling di openMP.

Elenco delle figure

Bibliografia

[Gus78] F.G. Gustavson. «two fast algorithms for sparse matrices: multiplication and permuted transposition». In: (1978).

[Pat80] S. V. Patankar. «Numerical Heat Transfer and Fluid Flow». In: (1980).

[Gol91] David Goldberg. «What Every Computer Scientist Should Know about Floating-Point Arithmetic». In: (1991).

[JRGS91] Cleve Moler John R. Gilbert e Robert Schreiber. «Sparse matrices in matlab: design and implementation». In: (1991).

[QV94] A. Quarteroni e A. Valli. «Numerical Approximation of Partial Differential Equations». In: (1994).

[GW97] R. A. Van De Geijn e J. Watts. «SUMMA: scalable universal matrix multiplication algorithm». In: (1997).

[Coh98] Cohen. «Structure prediction and computation of sparse matrix product». In: (1998).

[BC05] Daniel Bovet e Marco Cesati. «Understanding The Linux Kernel». In: (2005).

[MBK05] David Bommes Mario Botsch e Leif Kobbelt. «Efficient Linear System Solvers for Mesh Processing». In: (2005).

[Cor06] Jonathan Corbet. «Linux Weekly News:Trees II: red-black trees». In: (2006). URL: https://lwn.net/Articles/184495/.

[Lan07] Rob Landley. «Red-black Trees (rbtree) in Linux». In: (2007). URL: https://www.kernel.org/doc/Documentation/core-api/rbtree.rst.

[LeV07] R. J. LeVeque. «Finite Difference Methods for Ordinary and Partial Differential Equations». In: (2007).

[BG08] Aydın Buluc e John Gilbert. «New Ideas in Sparse Matrix Matrix Multiplication». In: (2008).

[MMAPD08] Narayanan Sundaram Jongsoo Park Michael J. Anderson Satya Gautam Vadlamudi Dipankar Das Sergey G. Pudov Vadim O. Pirogov Md. Mostofa Ali Patwary Nadathur Rajagopalan Satish e Pradeep Dubey. «Parallel Effcient Sparse Matrix-Matrix Multiplication on Multicore Platforms». In: (2008).

[Cor10] Jonathan Corbet. «Linux Weekly News:Augmented red-black trees». In: (2010). URL: https://lwn.net/Articles/388118/.

[BG12] Aydin Buluc e John R. Gilbert. «Parallel sparse matrix-matrix multiplication and indexing: implementation and experiments». In: (2012).

[FB12] Salvatore Filippone e Alfredo Buttari. «Object-Oriented Techniques for Sparse Matrix Computations in Fortran 2003». In: (2012).

[Ins12] American National Standards Institute. «NCITS/ISO/IEC 9899-2011 [2012] (ISO/IEC 9899-2011, IDT)». In: (2012).

[JPD15] Ulrike Meier Yang Dheevatsa Mudigere Jongsoo Park Mikhail Smelyanskiy e Pradeep Dubey. «High-Performance Algebraic Multigrid Solver Optimized for Multi-Core Based Distributed Parallel Systems». In: (2015).

[AAW16] Aydin Buluc James Demmel Laura Grigori Oded Schwartz Sivan Toledo Ariful Azad Grey Ballard e Samuel Williams. «Exploiting multiple levels of parallelism in sparse matrix-matrix multiplication». In: (2016).

[SF17] Davide Barbieri Alessandro Fanfarillo Salvatore Filippone Valeria Cardellini. «Sparse Matrix-Vector Multiplication on GPGPUs». In: (2017).

[XZ17] Jinchao Xu e Ludmil Zikatanov. «Algebraic Multigrid Methods». In: (2017).

[MM18] Malcom Cohen Michael Metcalf John Reid. «Modern Fortran explained incorporating ortran 2018». In: (2018).

[YN18] Ariful Azad Aydın Buluç Yusuke Nagasaka Satoshi Matsuoka. «HighPerformance Sparse MatrixMatrix Products on Intel KNL and Multicore Architectures». In: (2018).

[CH19] Israt Nisa Kunal Singh P. Sadayappan Changwan Hong Aravind Sukumaran-Rajam. «Adaptive Sparse Tiling for Sparse Matrix Multiplication». In: (2019).

[Cor19] Intel Corporation. «Intel® 64 and IA-32 Architectures Software Developer's Manual Volume 2 (2A, 2B, 2C & 2D): Instruction Set Reference, A-Z». In: (2019).

[Gao+19] Jianhua Gao et al. «A Systematic Survey of General Sparse Matrix-Matrix Multiplication». In: (2019).

[SF19] Buttari Alfredo Salvatore Filippone. «PSBLAS: a library for parallel linear algebra computation on sparse matrices». In: (2019).

[Boa20] OpenMP Architecture Review Board. «OpenMP Application Programming Interface». In: (2020).

[DA20] Gunduz Vehbi Demirci e Cevdet Aykana. «Cartesian Partitioning Models for 2D and 3D Parallel SpMM Algorithms». In: (2020).

[ZG20] David Edelsohn Ariful Azad Zhixiang Gu Jose Moreira. «Bandwidth Optimized Parallel Algorithms for Sparse Matrix-Matrix Multiplication using Propagation Blocking». In: (2020).

[DDF21] Pasqua D'Ambra, Fabio Durastante e Salvatore Filippone. «AMG Preconditioners for Linear Solvers towards Extreme Scale». In: (2021).

[For21] Message Passing Interface Forum. «MPI: A Message-Passing Interface Standard Version 4.0». In: (2021).

[RMS21] Zachary Weinberg Richard M. Stallman. «C PreProcessor Manual 11.2». In: (2021).

[SF21] Fabio Durastante Salvatore Filippone Pasqua D'Ambra. «AMG4PSBLAS User's and Reference Guide A guide for the Algebraic MultiGrid Preconditioners Package based on PSBLAS». In: (2021).

[SGDC21] Richard M. Stallman e the GCC Developer Community. «Using the GNU Compiler Collection, For gcc version 10.1.0». In: (2021).

[MPDT22] Wes McKinney e the Pandas Development Team. «pandas: powerful Python data analysis toolkit Release 1.4.2». In: (2022).

[Gmp] «GNU Multiple Precision Arithmetic Library». In: (n.d.). URL: https://gmplib.org/.