



UNIVERSITÀ
DELLA CALABRIA

DIPARTIMENTO DI INGEGNERIA INFORMATICA,
MODELLISTICA, ELETTRONICA E SISTEMISTICA (DIMES)

CORSO DI LAUREA MAGISTRALE IN INGEGNERIA
INFORMATICA

Relazione del Progetto:

PAGERANK

a cura di:

CLAUDIO BRANCACCIO (195299)

LEONARDO ZACCARO (195296)

A.A. 2017/2018

Descrizione del problema

Molti dei motori di ricerca odierni utilizzano una procedura basata su due fasi per recuperare pagine correlate alla ricerca di un utente. Nella prima fase viene eseguita l'elaborazione tradizionale del testo per trovare tutti i documenti utilizzando i termini della ricerca o, comunque, correlati ai termini della ricerca per il loro significato semantico. Con le enormi dimensioni del Web, questo primo passaggio può comportare il ritrovamento di migliaia di pagine correlate alla ricerca. Per rendere questa lista di pagine gestibile per un utente, molti motori di ricerca classificano questo elenco secondo un criterio di “ranking”. Uno dei modi più famosi per creare tale “ranking” è quello di sfruttare la struttura di collegamento ipertestuale del Web. Infatti il Web può essere strutturato come un grafo dove i nodi rappresentano le pagine che compongono la rete e gli archi i collegamenti che sussistono tra le pagine e per tale motivo si può così utilizzare l’analisi dei collegamenti come mezzo per classificare le pagine.

Uno dei sistemi di classificazione più efficaci basato sul collegamento è il PageRank, ossia il sistema utilizzato dal motore di ricerca di Google. Tuttavia, poiché il numero di pagine affini al Web è dell’ordine dei bilioni, il grafo che ne risulta è di dimensioni talmente elevate da poter pregiudicare le prestazioni dell’algoritmo PageRank.

Pertanto l’obiettivo del progetto è stato quello non solo di

fornire un’implementazione dell’algoritmo PageRank ma soprattutto di migliorarne le prestazioni utilizzando tecniche di ottimizzazione dell’hardware e strategie efficienti di memorizzazione di grandi quantità di dati al fine di fornire all’intera applicazione la scalabilità necessaria per poter eseguirlo su istanze di grandi dimensioni. L’ambiente sw/hw di riferimento è costituito dal linguaggio di programmazioni C (gcc), dal linguaggio assembly x86-32+SSE e dalla sua estensione x86-64+AVX (nasm) e dal sistema operativo Linux (Ubuntu).

Descrizione dell'algoritmo

La definizione di PageRank è fortemente legata al concetto di *eigenvector centrality*. In particolar modo il PageRank π_i di una pagina Web i è direttamente proporzionale al PageRank π_j di ogni suo incoming neighbour j ed inversamente proporzionale al numero di pagine d_j a cui j fa riferimento:

$$\pi_i = \sum_{j=1}^n \frac{a_{ji}}{d_j} \pi_j,$$

dove d_j denota il numero di archi uscenti da j (anche detto *outdegree*).

L'algoritmo PageRank lavora sul grafo rappresentante il Web utilizzando inizialmente la matrice di adiacenza ad esso afferente. Tale matrice, al fine di poter essere utilizzata nel calcolo effettivo del PageRank, subisce varie trasformazioni divenendo stocastica e irriducibile. In particolar modo occorre innanzitutto dividere ogni elemento appartenente alla matrice di adiacenza per l'outdegree corrispondente al nodo denotato dalla sua riga. Questa matrice (P) deve divenire una *matrice di transizione* facendo sì che la somma degli elementi appartenenti a ciascuna riga sia pari a 1. Per far ciò occorre sostituire la matrice P con la matrice P' , così definita

$$P' = P + D$$

con $D = \delta \cdot v$, dove $v = (1/n, 1/n, \dots, 1/n)^T$ e $\delta = (\delta_1, \delta_2, \dots, \delta_n)^T$, $\delta_i = 1$ se $d_i = 0$ e $\delta_i = 0$ se $d_i > 0$ ($1 \leq i \leq n$).

Un'altra proprietà da garantire è che il grafo di partenza sia fortemente connesso e per questo motivo la matrice P' viene sostituita dalla matrice P'' definita come segue

$$P'' = cP' + (1 - c)E$$

dove $E = u \cdot v$, con $u = (1, 1, \dots, 1)^T$ e $c \in [0, 1]$.

Una volta ottenuta tale matrice, per definire il PageRank, occorre calcolare l'autovettore principale ad essa associato, come segue

$$\pi = (P'')^T \pi$$

Per effettuare tale calcolo uno dei metodi più noti e più efficienti è il **metodo delle potenze**. Quest'ultimo è un metodo iterativo che, partendo da un'approssimazione iniziale $\pi^{(0)} = v$ computa ad ogni iterazione l'autovettore principale con approssimazione più accurata basandosi su quello calcolato nell'iterazione precedente come segue

$$\pi^{(k+1)} = (P'')^T \pi^{(k)}.$$

L'algoritmo termina quando la differenza $\Delta^{(k)} = \| \pi^{(k)} - \pi^{(k+1)} \|_1$ tra due soluzioni successive è minore di una soglia ϵ , restituendo come soluzione

$$\pi^* = \frac{\pi^{(k+1)}}{\| \pi^{(k+1)} \|_1}$$

Progettazione e implementazione dell'algoritmo in C

All'interno del progetto l'algoritmo PageRank è stato implementato a partire dal metodo delle potenze e lavora su due tipi di input distinti:

- *dense*, dove in tale caso l'algoritmo riceve in ingresso direttamente la matrice P'' .
- *sparse*, mentre in questo caso l'algoritmo riceve in ingresso il grafo rappresentante la rete come elenco di archi (i, j) .

Nel primo caso l'algoritmo esegue i calcoli utilizzando numeri floating-point solo a precisione doppia, mentre nel secondo caso viene gestita anche una codifica dei numeri floating-point a precisione singola. Inoltre è stata implementata una versione ottimizzata dell'algoritmo che mira ad accelerare la convergenza del metodo delle potenze sfruttando un ulteriore metodo, noto come **Power Extrapolation**.

Modalità dense

Per eseguire l'algoritmo secondo tale modalità è opportuno specificare il parametro *-dense* fornendo in input un file con

estensione *.matrix* contenente la matrice P'' .

L'algoritmo memorizza P'' non in una matrice esplicitamente ma bensì in un array di double al fine di evitare un numero eccessivo di indirizzamenti indiretti per accedere ai suoi elementi. Poiché tale struttura viene utilizzata dall'algoritmo soltanto per computare il prodotto $(P'')^T \pi^{(k)}$, allora la matrice P'' viene scandita per colonne. Per tale motivo è stato scelto un criterio di memorizzazione di tipo **column-major order** garantendo così l'accesso sequenziale agli elementi della matrice e di conseguenza la località spaziale massimizzando il numero di *cache hit*. La costruzione di tale struttura viene effettuata nel metodo *load_dense*.

Per quanto riguarda la modalità dense, l'implementazione dell'algoritmo vero e proprio è realizzata nel metodo *pagerank-FullNoOpt*. Per garantire una maggiore comprensione e modularità al codice, tale metodo è stato progettato come sequenza di chiamate a funzioni C, in particolare modo è costituito dalla chiamata alle seguenti funzioni:

- *prod*
- *differenceNormd*
- *assegnaD*
- *normalizzazione*

Il metodo *prod* si occupa della computazione del prodotto $(P'')^T \pi^{(k)}$ come segue

```

void prod(MATRIX P, VECTORD xk,int dim,int n,VECTORD xk1){
    int i,j;
    double somma;

    for(i=0;i<n;i++){
        somma=0.0;
        for(j=0;j<dim;j++){
            somma = somma + P[j+i*dim]*xk[j];
        }
        xk1[i]=somma;
    }
}

```

Come si evince da tale frammento di codice la matrice P'' è acceduta in maniera sequenziale per colonne e ciò motiva la scelta precedentemente illustrata di memorizzare tale struttura secondo il criterio del column-major order.

Il metodo *differenceNormd* si occupa invece di calcolare $\Delta^{(k)} = \|\pi^{(k)} - \pi^{(k+1)}\|_1$ per verificare che l'algoritmo sia giunto a convergenza tramite il confronto con la soglia di errore ϵ desiderata.

Al termine di ciascuna iterazione è necessario assegnare a $\pi^{(k)}$ l'autovettore calcolato nell'iterazione corrente, ossia $\pi^{(k+1)}$, per predisporre l'algoritmo all'esecuzione dell'iterazione successiva. Questa operazione viene effettuata dal metodo *assegnaD*.

Giunto a convergenza, l'algoritmo deve normalizzare l'autovettore calcolato durante l'ultima iterazione per ottenere la soluzione. Ciò viene effettuato nel metodo *normalizzazione*.

Modalità sparse

Per eseguire l'algoritmo secondo tale modalità è opportuno specificare il parametro *-sparse* fornendo in input un file con estensione *.graph* contenente la lista degli archi costituenti il grafo che rappresenta la rete. Tale lista di archi viene memorizzata

all'interno di un array di *location*, ossia uno *struct* rappresentante un generico arco. Questa memorizzazione viene effettuata nel metodo *load_sparse*.

Avendo come input solamente il grafo di partenza e non la matrice P'' , è opportuno esplicitare il prodotto $(P'')^T \pi^{(k)}$ come segue

$$\begin{aligned}\pi^{(k+1)} &= (P'')^T \pi^{(k)} = c(P')^T \pi^{(k)} + (1 - c)(ev)^T \pi^{(k)} \\ &= c(P')^T \pi^{(k)} + (1 - c)v^T \\ &= cP^T \pi^{(k)} + c(\delta v)^T \pi^{(k)} + (1 - c)v^T\end{aligned}$$

dal momento che $\pi^{(k)}$ è un vettore di probabilità e quindi $e^T \pi^{(k)} = 1$. Scritto in tale maniera diviene chiaro che il metodo delle potenze applicato a P'' può essere implementato con moltiplicazioni matrice-vettore sulla matrice P senza che P' e P'' siano esplicitamente formate o memorizzate. Il vantaggio che si ottiene da tale prodotto è che tale matrice è costituita da un elevato numero di elementi nulli in quanto ogni pagina ha nella realtà pratica un numero relativamente basso di collegamenti con le altre pagine. Ciò comporta che P sia una matrice sparsa e pertanto il prodotto matrice-vettore richiesto dal metodo delle potenze possa essere computato in $nnz(P)$ flops, dove $nnz(P)$ è il numero di elementi non nulli presenti in P .

Una volta effettuato il prodotto $cP^T \pi^{(k)}$ è possibile integrare il contributo di $c(\delta v)^T \pi^{(k)}$ e di $(1 - c)v^T$ senza che essi siano calcolati esplicitamente. Per farlo viene indicato con x il vettore risultante dal prodotto $(\delta v)^T \pi^{(k)}$, il quale è caratterizzato dal fatto che ciascuna sua componente sia pari a

$$\frac{\sum_{i|\delta_i=1} \pi_i^{(k)}}{n}$$

con n pari al numero dei nodi del grafo. Inoltre viene denotato con y il vettore risultante dal prodotto $cP^T\pi^{(k)}$. Sfruttando una delle proprietà del vettore $\pi^{(k+1)}$, ossia che la somma delle sue componenti è pari a 1 trattandosi di un vettore di probabilità, risulta che

$$\sum_{i=1}^n y_i + c \sum_{i=1}^n x_i + \frac{1-c}{n} \cdot n = 1$$

Da ciò si ha che

$$\sum_{i=1}^n y_i = c - c \sum_{i=1}^n x_i$$

Sottraendo ora ambo i membri ad 1 si ottiene

$$1 - \sum_{i=1}^n y_i = c \sum_{i=1}^n x_i + 1 - c$$

Dividendo per n ambo i membri si ottiene

$$\frac{1 - \sum_{i=1}^n y_i}{n} = \frac{c \sum_{i=1}^n x_i}{n} + \frac{1 - c}{n}$$

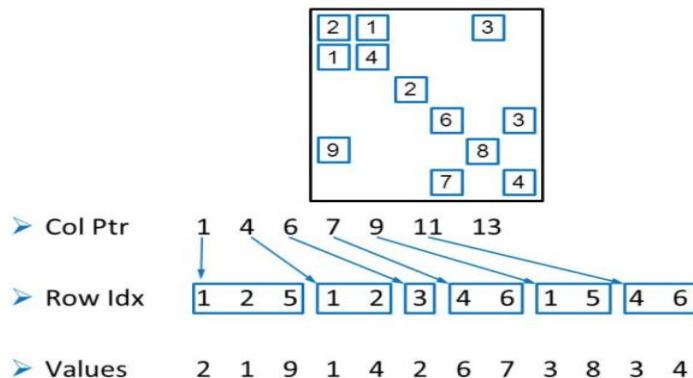
Poiché il secondo membro corrisponde a ciò bisogna sommare ad ogni componente di y per ottenere il vettore risultato $\pi^{(k+1)}$, è possibile dunque sfruttare tale relazione per evitare il calcolo del vettore x . Grazie a tale relazione è sufficiente pertanto sommare ad ogni componente di y il valore $\frac{1 - \sum_{i=1}^n y_i}{n}$ per ottenere il risultato finale.

Per sfruttare a pieno la sparsità della matrice P nel prodotto matrice-vettore è opportuno delineare la struttura adatta per la rappresentazione in memoria di tale matrice. La soluzione adottata all'interno di questo progetto è di rappresentarla utilizzando il formato **Compressed Sparse Column** (CSC).

Tale formato è costituito da 3 array:

- **values**, di lunghezza nnz e memorizza tutti i valori diversi da 0, ordinati da sopra a sotto e da sinistra a destra.
- **rowIdx**, di lunghezza nnz e che contiene l'indice di riga di ciascun elemento di *values*.
- **colPtr**, di lunghezza $n + 1$, dove $colPtr[i]$ contiene l'indice del primo elemento di *values* diverso da 0 della colonna i -esima. La lunghezza della colonna i -esima è determinata da $colPtr[i + 1] - colPtr[i]$, per questo motivo *column* deve essere di lunghezza $n + 1$.

Un esempio di memorizzazione di una matrice sparsa secondo il formato CSC è visibile nella seguente figura



Tuttavia ai fini del progetto è stato ritenuto opportuno introdurre una variante di tale rappresentazione. In particolare il vettore *values* è stato modificato nel seguente modo: sfruttando la proprietà che gli elementi di P' appartenenti alla stessa riga hanno il medesimo valore, è possibile comprimere la lunghezza di *values* da nnz a n . Nella posizione i -esima del vettore

values viene dunque memorizzato il valore che assume un qualunque elemento diverso da 0 della riga i-esima di P' , mentre in presenza di un *dangling node* il valore assunto da $values[i]$ è -1.

La costruzione della struttura appena esposta viene effettuata nel metodo *creaGF* nel caso in cui la modalità selezionata sia *single* mentre in *creaGD* se si lavora con modalità *double*.

Per quanto riguarda l'implementazione del metodo delle potenze, essa è realizzata nel metodo *pagerankSparseNoOptFloat* e nel metodo *pagerankSparseNoOptDouble* a seconda della precisione scelta. All'interno di tali metodi sono riutilizzati funzioni precedentemente descritte, quali *differenceNorm* e *normalizzazione* che possiedono gli stessi comportamenti analizzati nel caso dense. Sono presenti inoltre metodi aggiuntivi:

- *prodSparse*
- *somma*
- *completaSol*
- *assegnaSoluzione*
- *converti*

Ciascuna di queste funzioni presenta un'implementazione che utilizza calcoli con numeri a floating-point a precisione singola e una che invece utilizza calcoli con numeri a floating-point a precisione doppia.

Il metodo *prodSparse* calcola il prodotto matrice-vettore $cP^T\pi^{(k)}$ come segue

```

void prodSparseF(int* row, int* col, VECTORF vf, int N, VECTORF xk1, VECTORF xk, float*d,int M){
    int i,j,colel,currow=0;
    float s;
    for(i=0;i<M;i++){
        vf[i]= (xk1[i])*(d[i]);
    }
    colel=col[0];
    for(i=0;i<N;i++){
        s=0.0;
        for(j=0; j<colel; j++){
            s += vf[row[currow]];
            currow++;
        }
        xk[i]=s;
        colel=col[i+1]-col[i];
    }
}

```

Un primo accorgimento da fare è che l'elemento i -esimo del vettore $\pi^{(k)}$ viene moltiplicato sempre per un elemento della riga i -esima, la quale è caratterizzata dall'avere elementi con lo stesso valore. Questi valori, come prima puntualizzato, sono memorizzati nel vettore *values* che nel frammento di codice rappresentato in figura è indicato dal vettore *d*. Da ciò scaturisce che i prodotti parziali da sommare per ottenere gli elementi del vettore *y* siano sempre gli stessi mentre ciò che cambia sono i prodotti che devono essere sommati tra loro. Sfruttando tale proprietà è possibile, dunque, effettuare solo n prodotti parziali anziché nnz . Come si evince dalla figura il primo ciclo si occupa di calcolare i prodotti parziali e di assegnarli al vettore *vf*, mentre il secondo ciclo calcola elemento per elemento il vettore *y* sommando i prodotti parziali corrispondenti alla riga dove si trovano gli elementi non nulli.

Una volta calcolato tale prodotto occorre integrarlo con i contributi necessari per ottenere il vettore $\pi^{(k+1)}$. Come detto in precedenza occorre dunque calcolare la quantità $\sum_{i=1}^n y_i$ per poi sommare a ciascuna componente di *y* la quantità $\frac{1-\sum_{i=1}^n y_i}{n}$. Queste operazioni vengono effettuate rispettivamente nei metodi

somma e *completaSol*.

Il metodo *assegnaSoluzione*, oltre a svolgere la stessa funzionalità del metodo *assegna* utilizzato nella modalità dense, azzerà il vettore contenente l'autovettore calcolato nell'iterazione corrente.

Infine il metodo *converti* viene utilizzato dalla modalità single per restituire il PageRank con numeri floating-point a precisione doppia.

Modalità opt

I metodi fino ad ora descritti implementano l'algoritmo PageRank utilizzando il metodo delle potenze standard. Esistono però diversi metodi risolutivi che permettono di accelerare la convergenza del metodo delle potenze. Tali metodi possono essere divisi in due classi:

- quelli che risparmiano tempo riducendo il carico di lavoro di ogni iterazione.
- quelli che puntano a ridurre il numero totale di iterazioni.

Soltamente però ad iterazioni più leggere corrispondono un maggior numero di passi per raggiungere la convergenza, mentre un numero di iterazioni minore corrisponde un maggior carico di lavoro per ciascun passo dell'algoritmo.

Un esempio di metodo appartenente alla prima classe è l'**Adaptive PageRank**. Quest'ultimo riduce il lavoro di ogni iterazione osservando in maniera più dettagliata gli elementi contenuti nel vettore soluzione corrente. In particolar modo nota quali pagine convergano al loro PageRank più velocemente rispetto alle altre pagine. Quando gli elementi del vettore PageRank convergono, l'Adaptive PageRank non li prende più in considerazione

nei calcoli effettuati nelle iterazioni successive. Sebbene da un lato tale metodo consenta un'accelerazione del metodo delle potenze, dall'altro risulta essere oneroso a causa della continua riorganizzazione della struttura contenuta la matrice.

All'interno di questo progetto si è scelto di implementare un metodo di risoluzione facente parte della seconda classe, noto come **Power Extrapolation**¹. Tale metodo assume che $\pi^{(k-d)}$ possa essere espresso come combinazione lineare degli autovettori $\{u_1, u_2, \dots, u_{d+1}\}$, dove gli autovalori di $\{u_2, \dots, u_{d+1}\}$ sono la radice di d -esima dell'unità, scalata di c .

$$\pi^{(k-d)} = u_1 + \sum_{i=2}^{d+1} \alpha_i u_i$$

Poiché il metodo delle potenze genera le iterazioni attraverso la moltiplicazione successiva per $(P'')^T$, si può scrivere $\pi^{(k+1)}$ come

$$\begin{aligned} \pi^{(k+1)} &= ((P'')^T)^d \pi^{(k-d)} \\ &= ((P'')^T)^d (u_1 + \sum_{i=2}^{d+1} \alpha_i u_i) \\ &= u_1 + \sum_{i=2}^{d+1} \alpha_i \lambda_i^d u_i \end{aligned}$$

Ma dal momento che $\lambda_i = cd_i$, dove d_i è la radice d -esima dell'unità,

$$\pi^{(k+1)} = u_1 + c^d \sum_{i=2}^{d+1} \alpha_i u_i$$

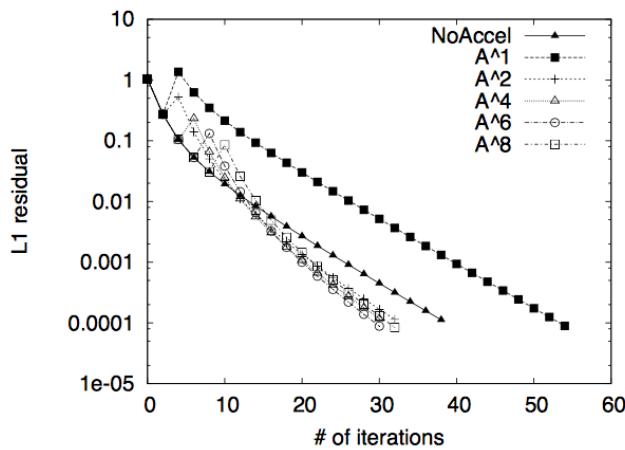
¹Per la descrizione e la valutazione delle prestazioni del Power Extrapolation è stato consultato il seguente articolo <http://ilpubs.stanford.edu:8090/605/1/2003-45.pdf>

Ciò permette di ottenere u_1 come segue

$$u_1 = \frac{\pi^{(k+1)} - c^d \pi^{(k-d)}}{1 - c^d}$$

Gli autovalori considerati sono di modulo c e derivano dai nodi foglia del grafo fortemente connesso del Web che costituiscono cicli di lunghezza d . Poiché si sa empiricamente che il Web possiede tali nodi foglia, è probabile che eliminare l'errore lungo le dimensioni degli autovettori corrispondenti a questi autovalori accelera la convergenza del PageRank.

Per via sperimentale si è analizzato come il valore di d corrispondente a 6 sia ottimale per accelerare la convergenza dell'algoritmo per grandi istanze. In particolar modo per istanze piccole si ha un lieve calo delle prestazioni dovuto alla presenza di calcoli aggiuntivi, ma con un numero di iterazioni superiori a 15 si ottiene uno speed-up, che è massimo per d . Al variare di d è possibile vedere come variano le prestazioni del Power Extrapolation



Infine una soluzione alternativa per computare il vettore $\pi^{(k+1)}$ è quella di calcolare il prodotto $cP^T\pi^{(k)}$ ottenendo il vettore y

e di sommare a ciascuna delle componenti di y il valore $\frac{1-c}{N}$. In tal modo si assume implicitamente l'assenza di *dangling nodes* ottenendo una convergenza più veloce del metodo delle potenze. Tuttavia tale assunzione comporta una approssimazione del risultato finale.

Implementazione dell'algoritmo in linguaggio Assembly

Il codice fino ad ora illustrato è stato tradotto in linguaggio Assembly fornendo due soluzioni software, una per l'architettura x86-32+SSE e l'altra per l'architettura x86-64+AVX. Nell'implementazione di tali soluzioni sono state adottate due tecniche di ottimizzazione:

- *loop unrolling*
- *code vectorization*

Il **loop unrolling** consiste nello srotolamento di un ciclo riscrivendolo come una sequenza ripetuta di istruzioni simili e indipendenti tra loro. Ciò comporta la diminuzione di controlli che vengono effettuati normalmente all'inizio di ogni iterazione con la conseguente diminuzione dei salti condizionati. Inoltre questa tecnica può essere utilizzata per rendere i programmi più facilmente parallelizzabili.

Il *loop unrolling* può essere combinata con la tecnica del **code vectorization**. Tale tecnica risulta essere molto importante per aumentare il grado di parallelismo del codice, dando la possibilità di effettuare più operazioni dello stesso tipo contemporaneamente. L'idea che sta dietro a tale tecnica è quella di

sfruttare l'adiacenza di dati con proprietà simili in maniera tale da poter gestire un gruppo di dati anziché un singolo elemento.

Un'altra tecnica di ottimizzazione è il **cache blocking** che sfrutta l'organizzazione hardware della cache suddividendo logicamente in blocchi la struttura dati, caricando poi in cache un blocco alla volta per poi utilizzarlo in elaborazioni multiple. Attraverso l'utilizzo di tale tecnica si garantisce la località temporale grazie al trasferimento in cache degli elementi costituenti il blocco. Tuttavia, poiché nel prodotto matrice-vettore gli elementi della matrice vengono acceduti un'unica volta durante il calcolo dello stesso, non è necessario garantire la località temporale e pertanto la tecnica del *cache blocking* non è stata implementata.

Il *loop unrolling* e il *code vectorization* sono stati implementati in linguaggio Assembly relativo sia all'architettura x86-32+SSE e sia all'architettura x86-64+AVX. In particolar modo per implementare la tecnica del *loop unrolling* sono stati sfruttati i registri vettoriali a virgola mobile messi a disposizione da ciascuna architettura ovvero gli 8 registri XMM a 128 bit per quanto riguarda SSE e i 16 registri YMM a 256 bit per quanto riguarda AVX.

Utilizzando quindi tali registri per conservare i dati ed effettuare le operazioni, è stato possibile introdurre due fattori di unrolling, uno pari a 4 e uno pari ad 8, per quanto riguarda l'architettura SSE successivamente aumentati a 8 e 16 in AVX grazie alla presenza del maggior numero di registri.

L'implementazione della tecnica del *code vectorization* è stata invece resa possibile per l'esistenza di istruzioni tipo *packed* grazie alle quali si ha la possibilità di effettuare operazioni dello stesso tipo in parallelo sui gruppi di numeri contenuti nei regi-

stri vettoriali. In particolar modo un registro XMM, essendo a 128 bit, può contenere 2 elementi di tipo double o 4 di tipo float con i quali è possibile effettuare istruzioni in parallelo mediante un'istruzione *packed*, mentre i registri YMM possono contenere 4 elementi double e 8 di tipo float essendo a 256 bit aumentando ancora di più il grado di parallelismo. Per poter utilizzare le istruzioni di tipo *packed* in maniera efficiente è stato opportuno istanziare le strutture dati in maniera allineata utilizzando l'istruzione `_mm_malloc`.

Combinando la tecnica del *code vectorization* e del *loop unrolling* è stato possibile dunque eseguire:

- 8 o 16 operazioni in una singola iterazione di un ciclo per quanto riguarda l'architettura SSE con numeri a precisione doppia
- 16 o 32 operazioni in una singola iterazione di un ciclo per quanto riguarda l'architettura SSE con numeri a precisione singola
- 32 o 64 operazioni in una singola iterazione di un ciclo per quanto riguarda l'architettura AVX con numeri a precisione singola
- 64 o 128 operazioni in una singola iterazione di un ciclo per quanto riguarda l'architettura AVX con numeri a precisione singola

Prima di poter utilizzare tali tecniche all'interno del progetto, è stato però necessario predisporre le strutture dati in maniera tale da poter evitare dei controlli troppo dispendiosi sulle loro dimensioni che devono per forza di cose essere multiple del numero di operazioni effettuate all'interno di una singola iterazione

del ciclo. Per fare ciò è stata utilizzata la tecnica del **padding**, ovvero sono state allocate aree di memoria in più contenenti degli 0 in ogni struttura in modo da rendere le dimensioni di tali strutture multiple della fattore desiderato. In particolare sono stati effettuati 2 tipi di *padding* per inizializzare le strutture a seconda del fattore di unrolling utilizzato dai metodi che richiamano tali strutture. Per farlo vengono calcolate due variabili nei metodi *load* dette *restoN* e *restoM* che indicano il numero di 0 da aggiungere a ciascuna struttura. Un esempio è il seguente

```

restoN = 8 - cols%8;
if(restoN==8) restoN=0;
restoM = 16 - cols%16;
if(restoM==16) restoM=0;

MATRIX data = alloc_matrix(rows,cols+restoN);
for(i=0;i<rows;i++){
    for(j=0;j<cols;j++){
        status = fread(data+i+j*(rows+restoN), sizeof(double), 1, fp);
    }
}

for(i=rows;i<rows+restoN;i++){
    for(j=0;j<cols;j++) data[i+j*(rows+restoN)]=0.0;
}

```

Come visibile nel metodo *load_dense* viene effettuato un *padding* per ciascuna colonna della matrice poiché all'interno del metodo *prod* vengono applicati loop unrolling e code vectorization nel ciclo interno che itera sugli elementi di ciascuna colonna. Poiché tali tecniche non vengono applicate nell'iterazione sulle righe è stato ritenuto opportuno non introdurre ulteriori 0 sulle righe così da evitare un *padding* eccessivo che potrebbe comportare un calo delle prestazioni.

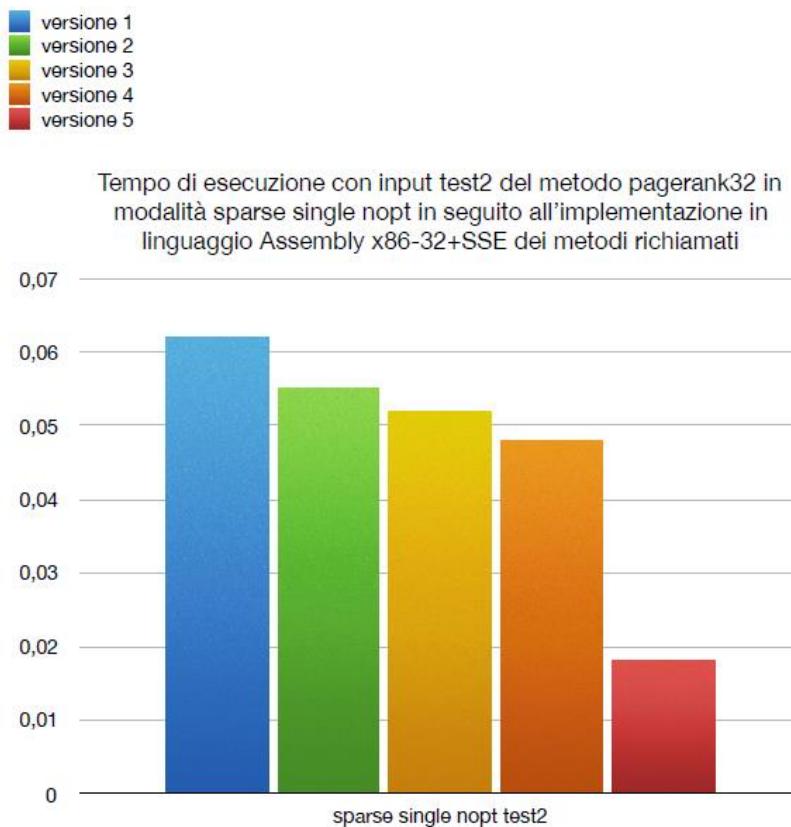
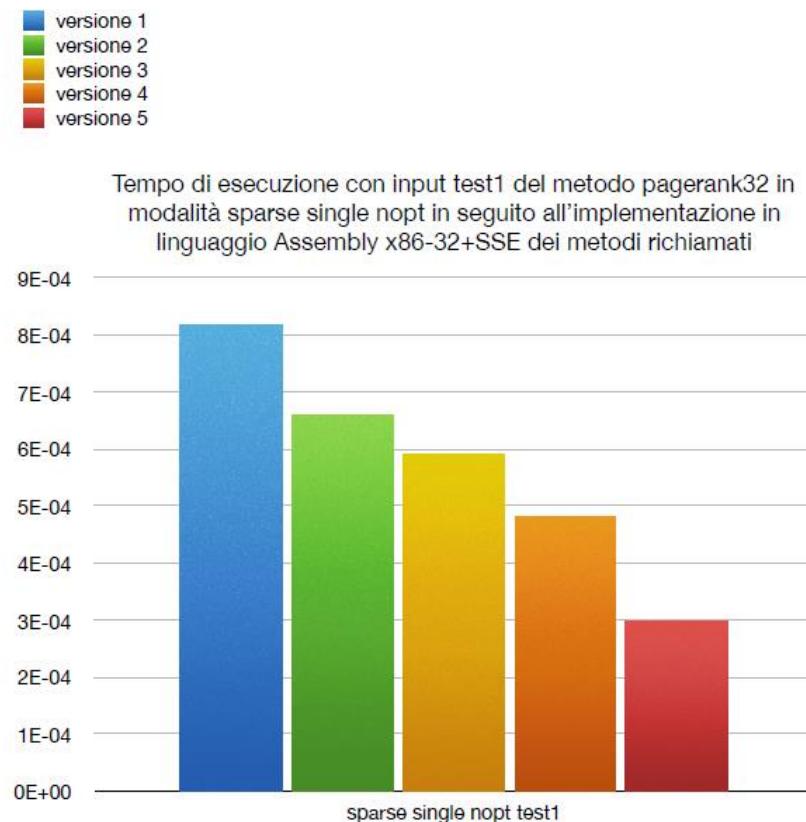
Dopo le precedenti accortezze è stato dunque possibile tradurre in linguaggio Assembly funzione per funzione l'algoritmo in C esposto precedentemente. Le istruzioni da utilizzare all'in-

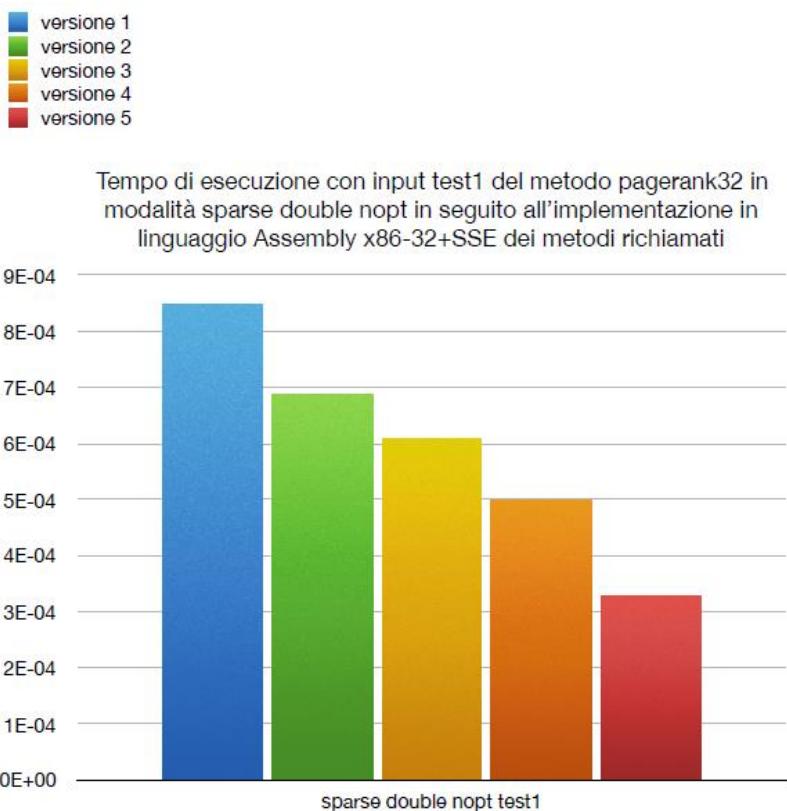
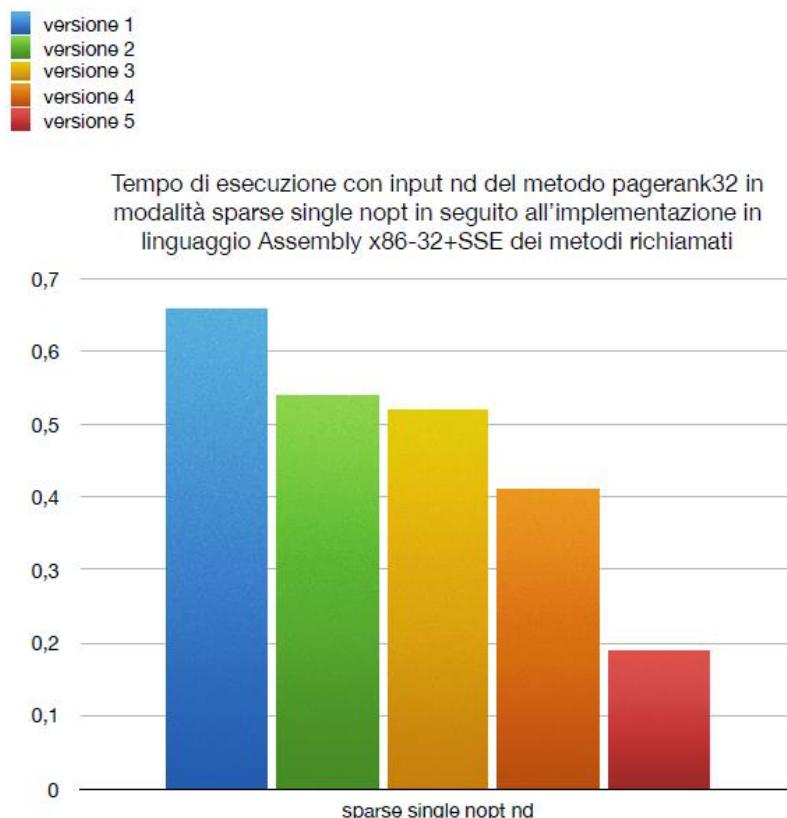
terno delle varie funzioni Assembly sono state scelte in modo accurato per ridurre il tempo totale di esecuzione e tenendo in considerazione la latenza di ciascuna di esse. In particolare modo laddove il divisore sia un parametro noto, come all'interno di *prodSparse*, è possibile passare come parametro il reciproco del divisore al fine di utilizzare l'istruzione *mulps* o *mulpd* (entrambe con latenza pari a 5;4) a seconda della precisione scelta al posto di *divps* (con latenza pari a 27;39) o *divpd* (con latenza pari a 27;69). Invece nel caso delle funzioni tradotte in linguaggio Assembly con l'architettura x86-64+AVX per copiare in tutti i campi di un registri il valori contenuto in uno di essi è stata utilizzata l'istruzione *vshuf* (con latenza pari a 1) che seppur più complessa risulta essere più veloce dell'istruzione *vpermil* (con latenza pari a 3). Oppure, sempre nel caso di funzioni tradotte in linguaggio Assembly con l'architettura x86-64+AVX, per evitare un calo di prestazioni per il passaggio da una modalità all'altra sono state utilizzate unicamente le istruzioni che lavorano a 256 bit, ossia le istruzioni con il prefisso *v-*, senza mai alternarle a quelle che lavorano a 128 bit anche laddove queste ultime risultavano essere più convenienti come nel caso del prodotto sparso. In quest'ultimo caso infatti le *v-instruction* risultano essere opportune nel calcolo dei prodotti parziali ma non nel calcolo del prodotto effettivo poiché avviene in modo scalare calcolando un prodotto ad ogni iterazione; tuttavia per non compromettere le prestazioni, anche in quest'ultimo caso sono state utilizzate le *v-instruction*. Un altro esempio si ha nel calcolo della norma, in particolare modo nel calcolo del valore assoluto di un numero, dove al posto di utilizzare un approccio basato sulle maschere dispendioso sia per l'utilizzo eccessivo dei registri sia per la duplicazione delle operazioni di somma, una

per i valori positivi e una per i negativi, si è preferito utilizzare le istruzioni *psll* e *psrl* per azzerare il bit segno di ogni valore floating-point ottenendone così il modulo così da poter utilizzare un'operazione di somma unica.

Nelle seguenti figure sono riportati i tempi di convergenza delle varie versioni dell'algoritmo implementato in linguaggio Assembly con l'architettura **x86-32+SSE** eseguito su una macchina virtuale con sistema operativo *Ubuntu* utilizzando 3 tipi di input ovvero:

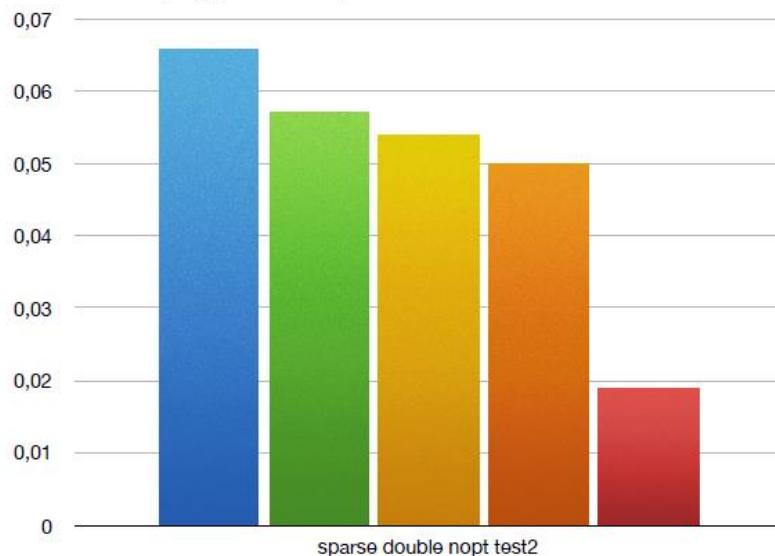
- **Test1:** grafo contenente 2319 nodi e 5000 archi nel caso sparse, mentre 5377761 archi nel caso dense.
- **Test2:** grafo contenente 12008 nodi e 237010 archi nel caso sparse, mentre 44192064 archi nel caso dense.
- **NotreDam (nd):** grafo con rappresentazione solo sparsa contenente 325729 nodi e 1469679 archi.





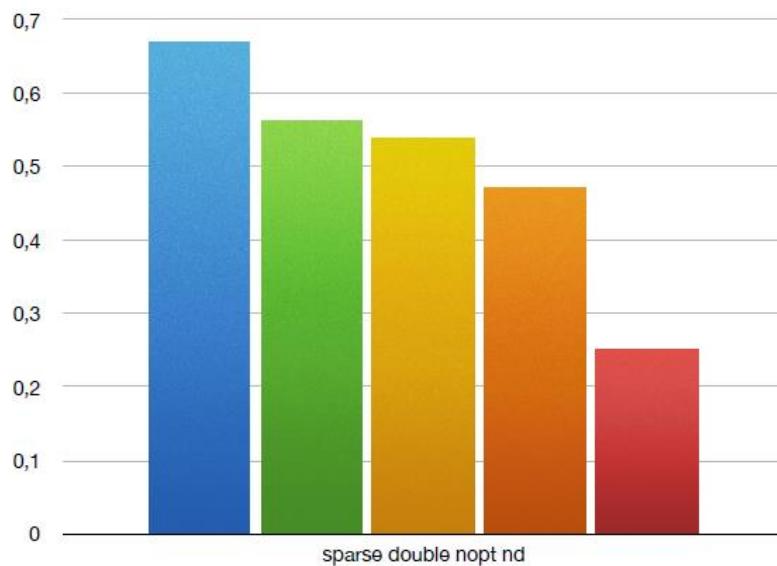
█ versione 1
█ versione 2
█ versione 3
█ versione 4
█ versione 5

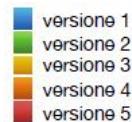
Tempo di esecuzione con input test2 del metodo pagerank32 in modalità sparse double nopt in seguito all'implementazione in linguaggio Assembly x86-32+SSE dei metodi richiamati



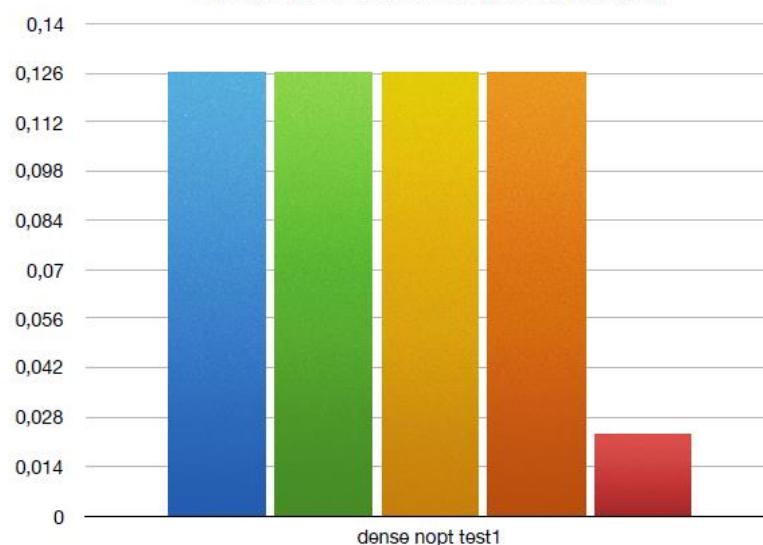
█ versione 1
█ versione 2
█ versione 3
█ versione 4
█ versione 5

Tempo di esecuzione con input nd del metodo pagerank32 in modalità sparse double nopt in seguito all'implementazione in linguaggio Assembly x86-32+SSE dei metodi richiamati

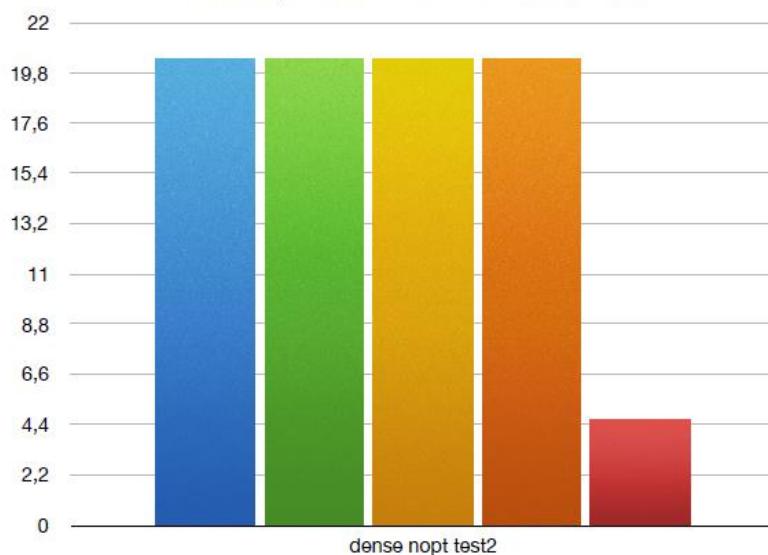


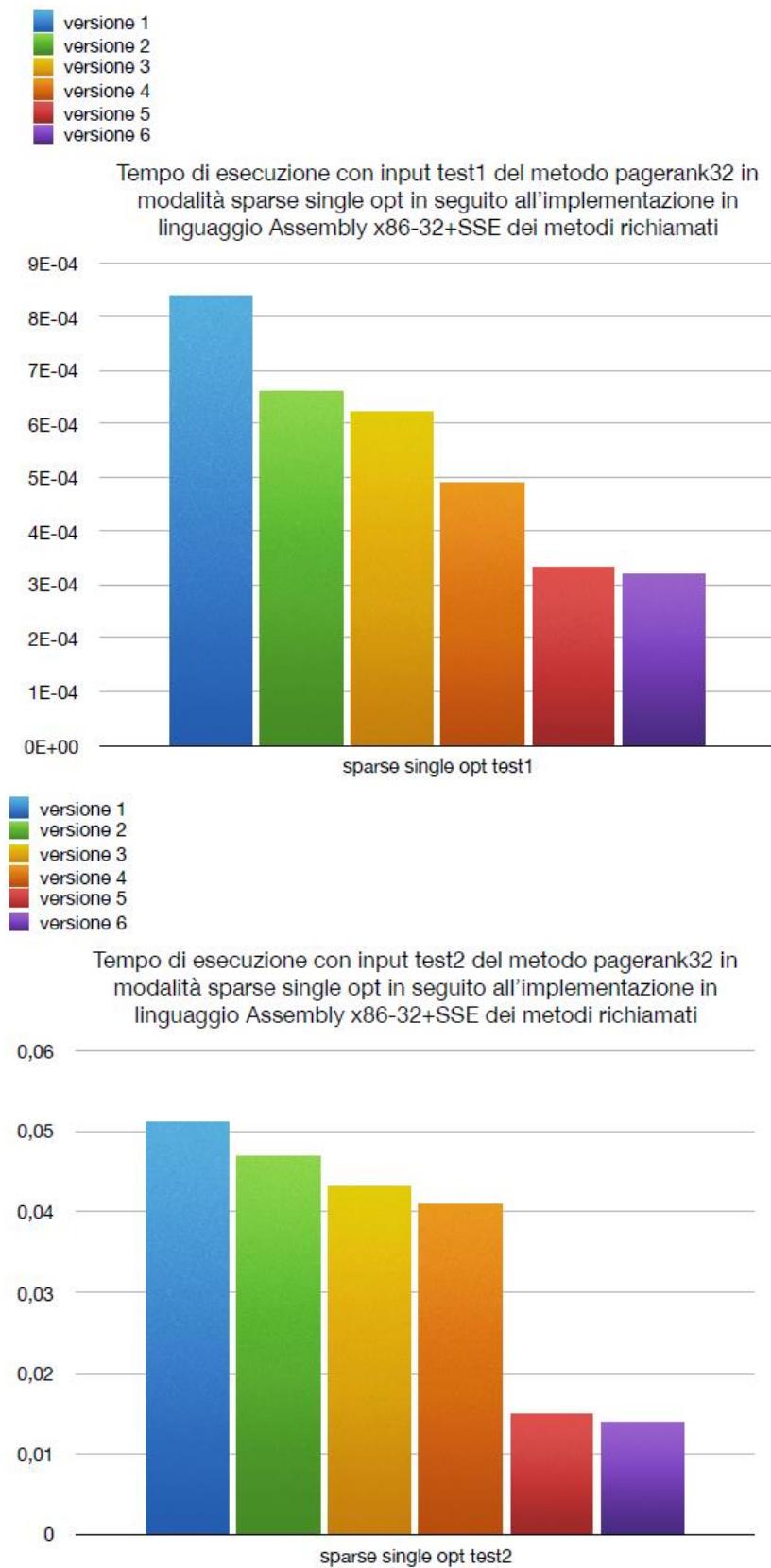


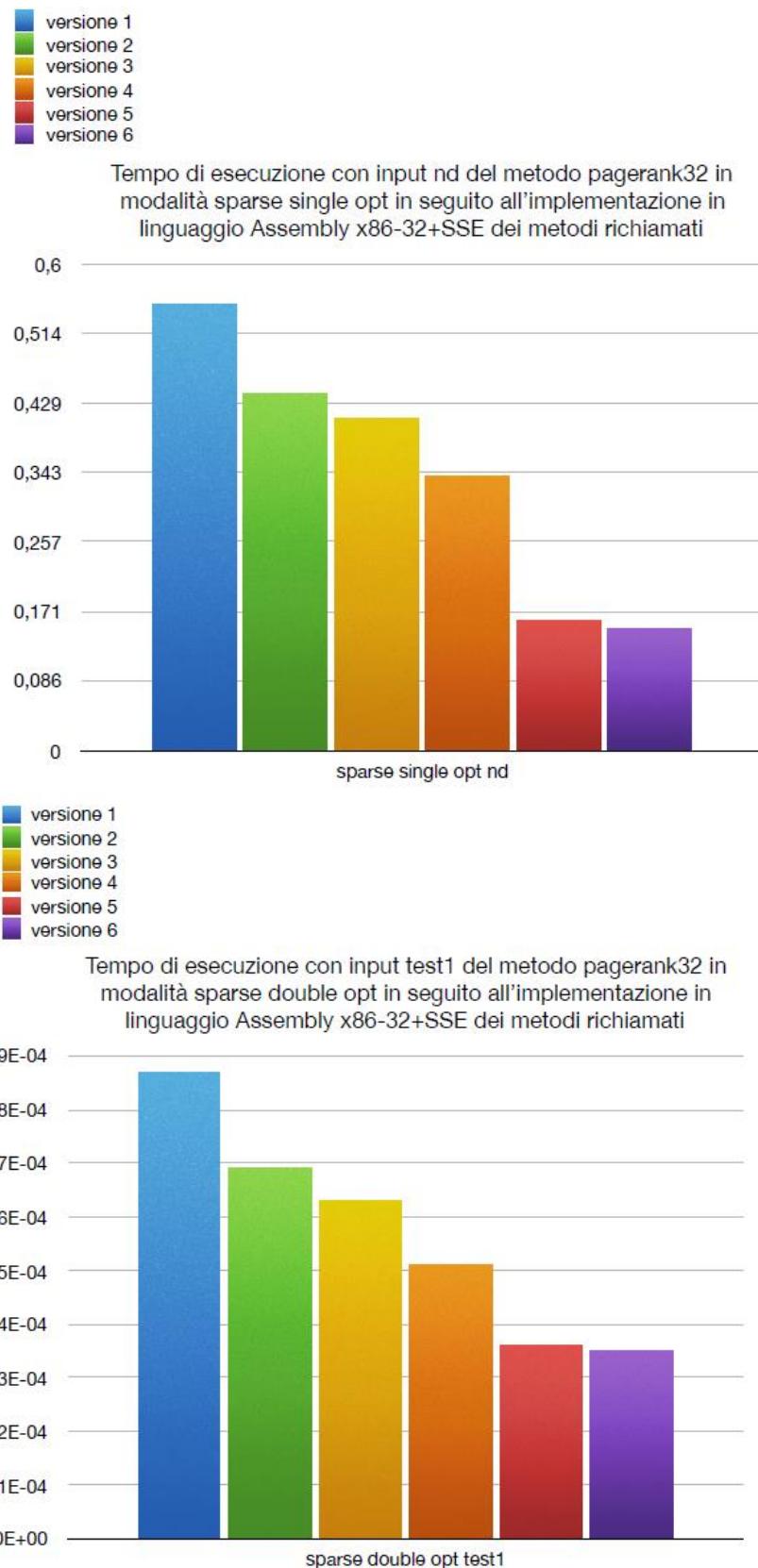
Tempo di esecuzione con input test1 del metodo pagerank32 in modalità dense nopt in seguito all'implementazione in linguaggio Assembly x86-32+SSE dei metodi richiamati

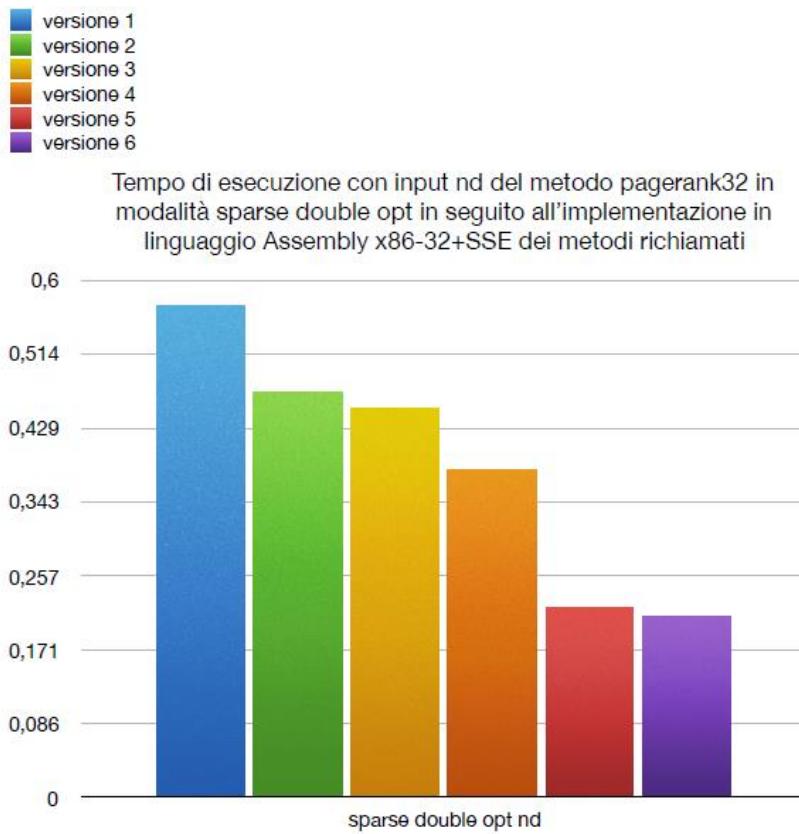
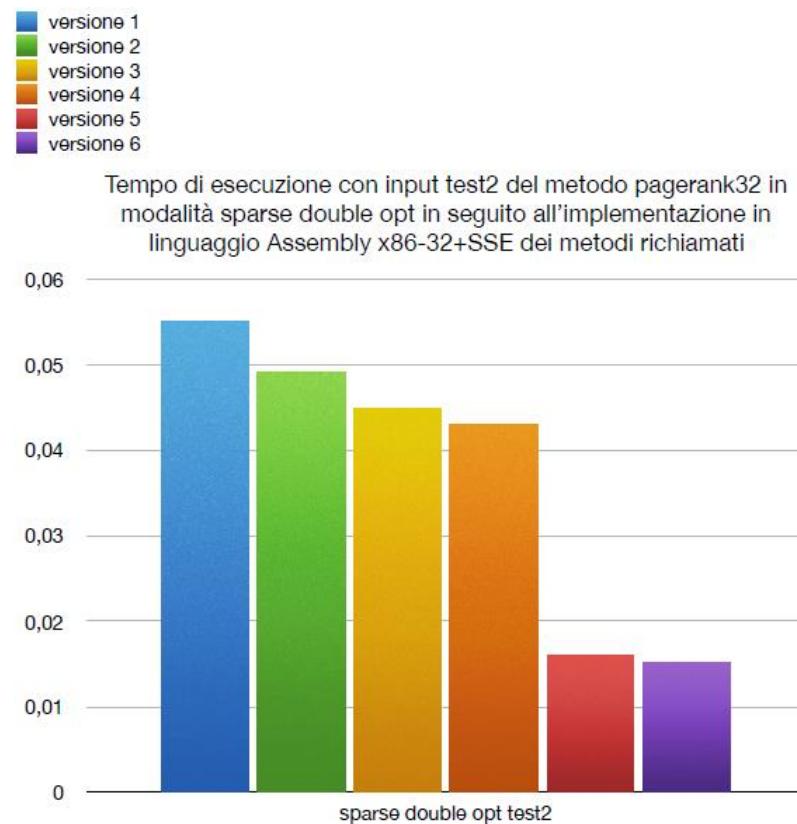


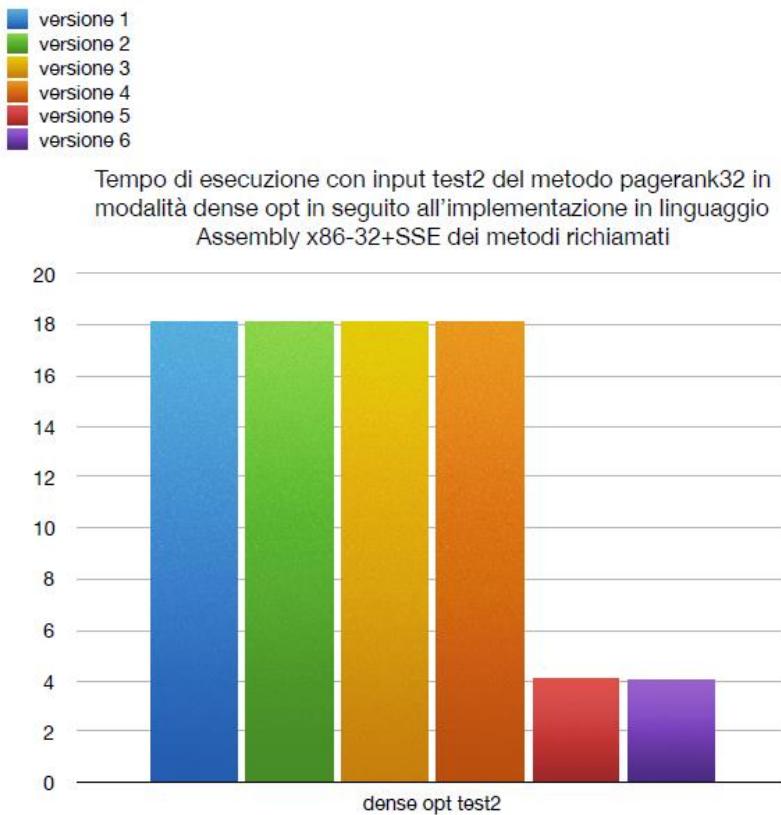
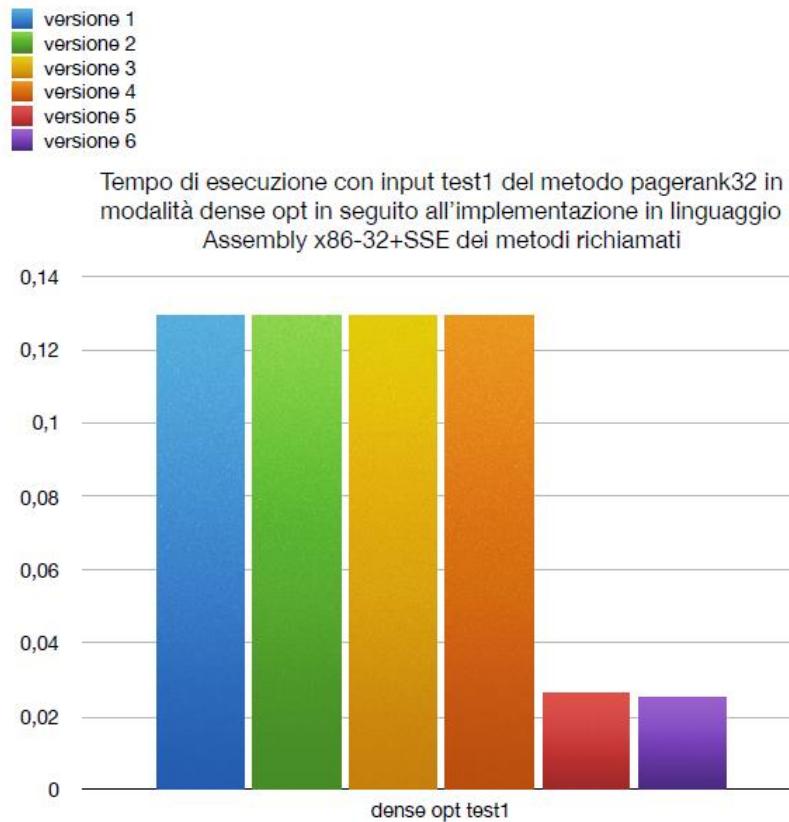
Tempo di esecuzione con input test2 del metodo pagerank32 in modalità dense nopt in seguito all'implementazione in linguaggio Assembly x86-32+SSE dei metodi richiamati











I grafici sopra mostrati evidenziano l'andamento delle prestazioni al variare delle versioni. In particolar modo le varie versioni dell'algoritmo sono le seguenti:

- **Versione1:** Algoritmo implementato unicamente in C.
- **Versione2:** Algoritmo con funzione *differenceNorm* tradotta in linguaggio Assembly.
- **Versione3:** Algoritmo con funzioni *differenceNorm* e *assegna* tradotte in linguaggio Assembly.
- **Versione4:** Algoritmo con funzioni *differenceNorm*, *assegna*, *completaSol* e *somma* tradotte in linguaggio Assembly.
- **Versione5:** Algoritmo con funzioni *differenceNorm*, *assegna*, *completaSol*, *somma* e *prod* tradotte in linguaggio Assembly.
- **Versione6:** Algoritmo con funzioni *differenceNorm*, *assegna*, *completaSol*, *somma*, *prod* e *powerExtr* tradotte in linguaggio Assembly. Tale versione è presente solo nel caso *opt*.

I tempi mostrati evidenziano dunque le migliori che ogni funzione tradotta in Assembly porta alle prestazioni dell'algoritmo grazie all'aumento del grado di parallelismo dovuto all'utilizzo delle istruzioni vettoriali.

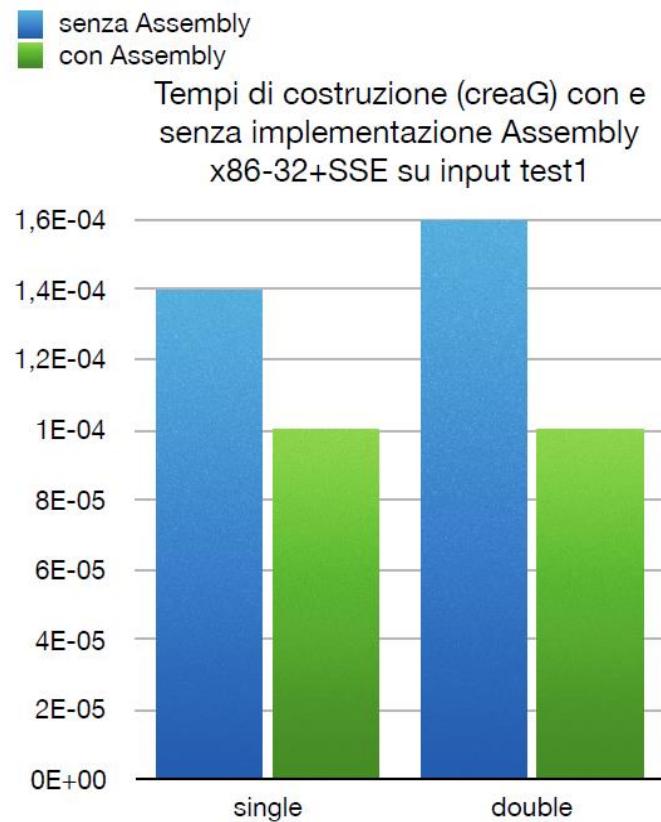
La versione dense non trae grossi benefici dalla traduzione in Assembly delle prime funzioni in quanto il tempo che caratterizza l'esecuzione di tale algoritmo dipende soprattutto dal prodotto matrice-vettore. Nel caso dense si può notare infatti come in qualsiasi versione e con qualsiasi input ci sia uno speed-up notevole delle prestazioni nella versione 5, ovvero quando il prodotto viene tradotto in Assembly e dunque parallelizzato.

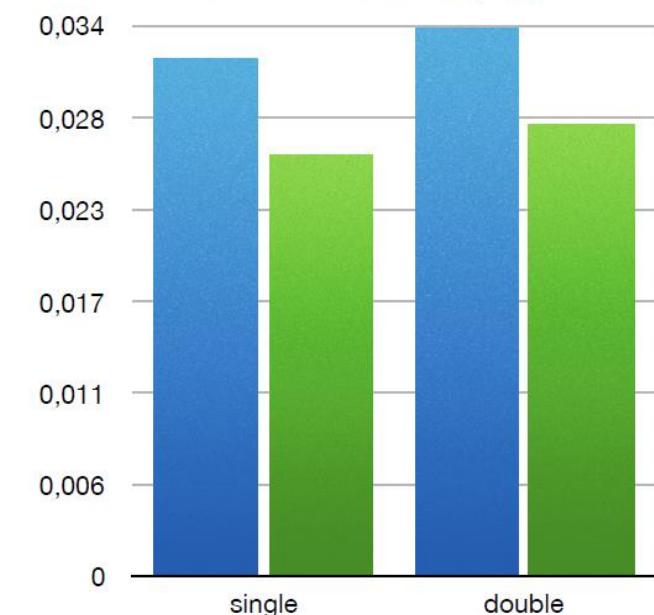
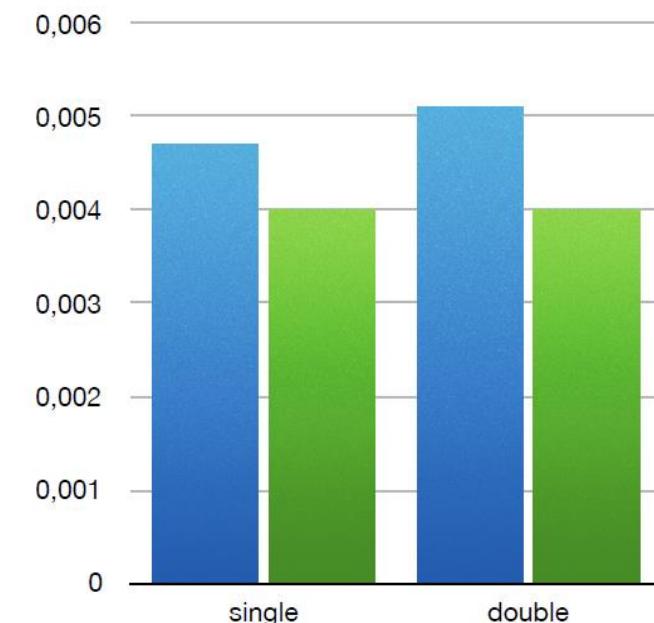
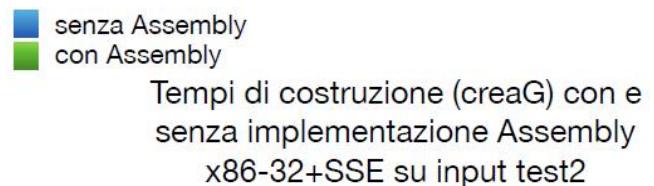
Al contrario invece, il caso sparse presenta già in C una versione del prodotto molto più veloce rispetto al caso dense grazie alla sua rappresentazione che consente di effettuare un numero di operazioni proporzionale solo agli elementi diversi da 0, che nella realtà pratica solo molto minori di n^2 . Questo consente dunque di poter meglio analizzare i benefici portati dalla traduzione delle funzioni diverse dal prodotto e pertanto si può osservare un miglioramento delle prestazioni già dalla versione 2. Sebbene la struttura utilizzata consenta di realizzare un prodotto molto più efficiente, questo risulta però difficilmente parallelizzabile a causa della variazione del numero di elementi presenti sulle colonne della matrice rappresentata che hanno delle dimensione molto ridotte, con il conseguente aumento del numero di accessi indiretti che incidono fortemente sulle prestazioni. Inoltre in questo caso l'attuazione della tecnica del *padding* risulterebbe particolarmente dispendiosa in quanto il numero 0 introdotto sarebbe uguale se non superiore al numero di elementi diversi da 0 della matrice. A causa di tali considerazioni, il prodotto sparso è stato dunque tradotto in maniera scalare, con l'unica eccezione del calcolo dei prodotti parziali, computati invece seguendo una logica vettoriale.

L'ultima cosa da notare è infine il miglioramento delle prestazioni nel caso *opt*, che si può notare soprattutto con input di taglia più grande o, comunque, con istanze che fanno convergere il metodo delle potenze con un numero di iterazioni superiore a 15 come *test2* ed *nd*. Viceversa con *test1* vi assiste ad un leggero calo delle prestazioni in quanto le iterazioni effettuate sono soltanto 7.

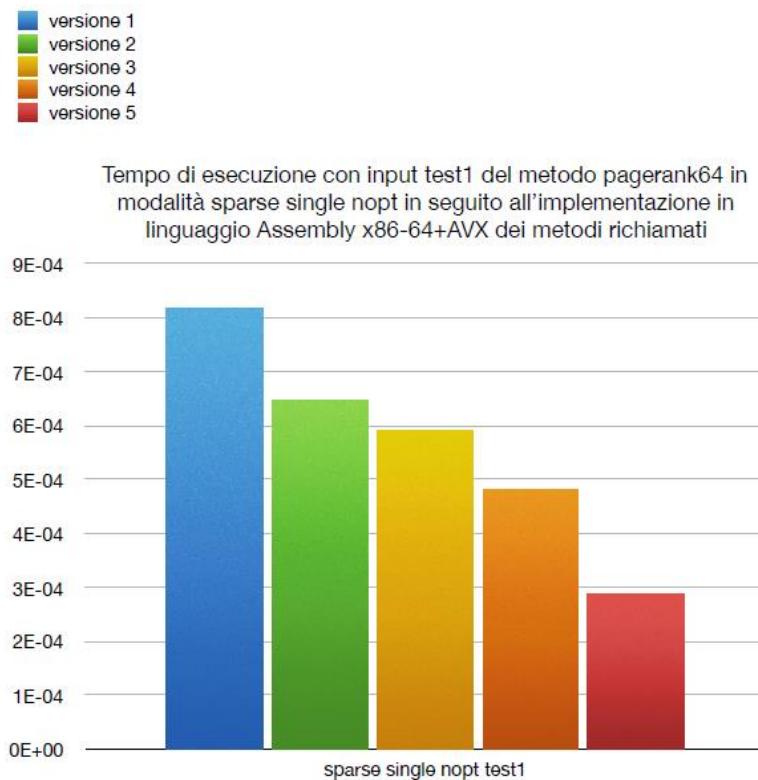
Nel caso sparse prima di poter lanciare l'algoritmo occorre costruire la struttura **CSC**, illustrata in precedenza, attraverso

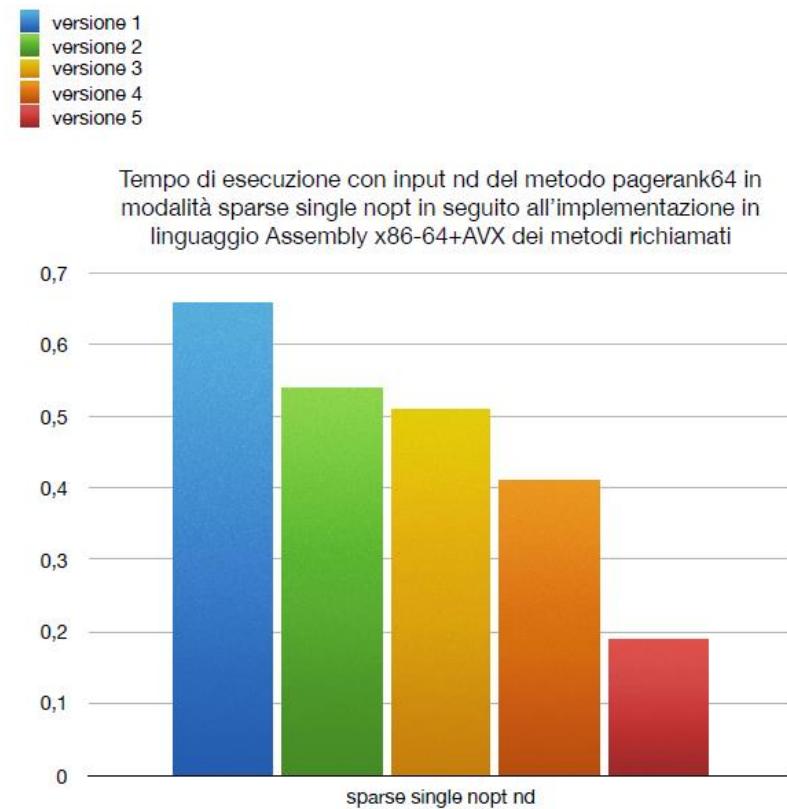
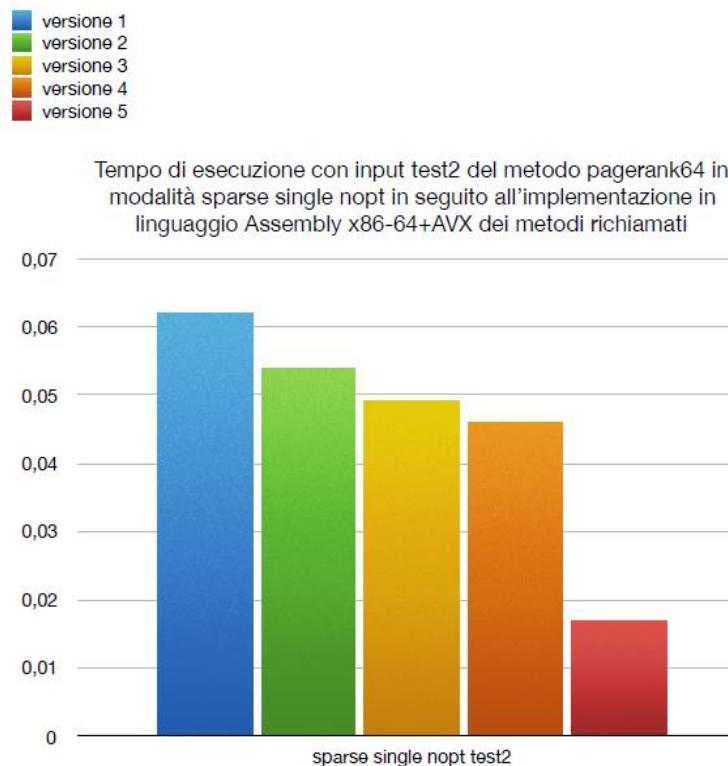
il metodo *creaG* il quale è stato anch'esso tradotto in linguaggio assembly. Di seguito è possibile vedere come la costruzione sia velocizzata grazie alla traduzione di parte del metodo con l'architettura **x86-32+SSE**.

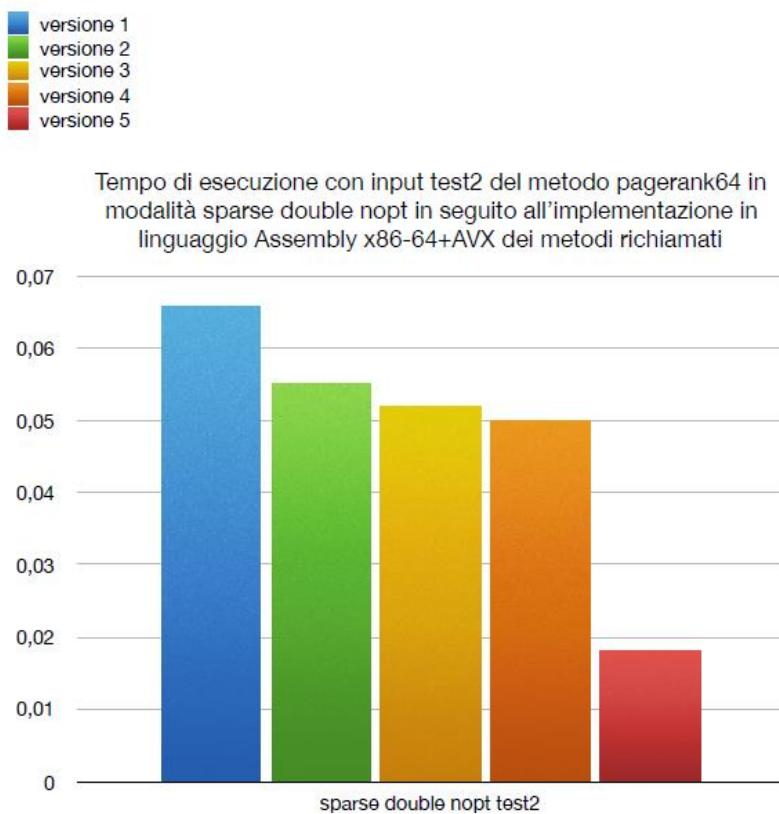
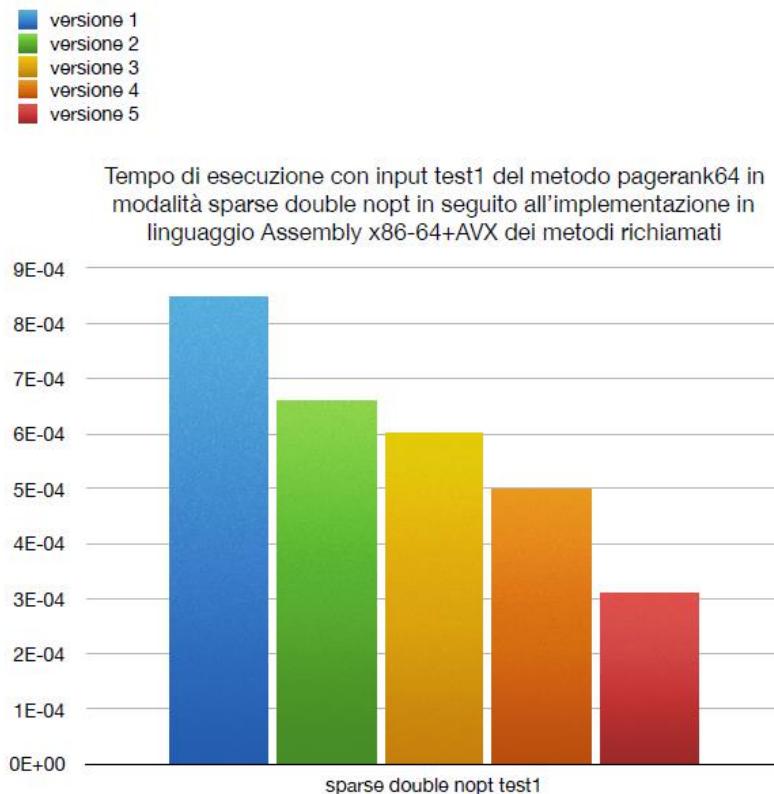


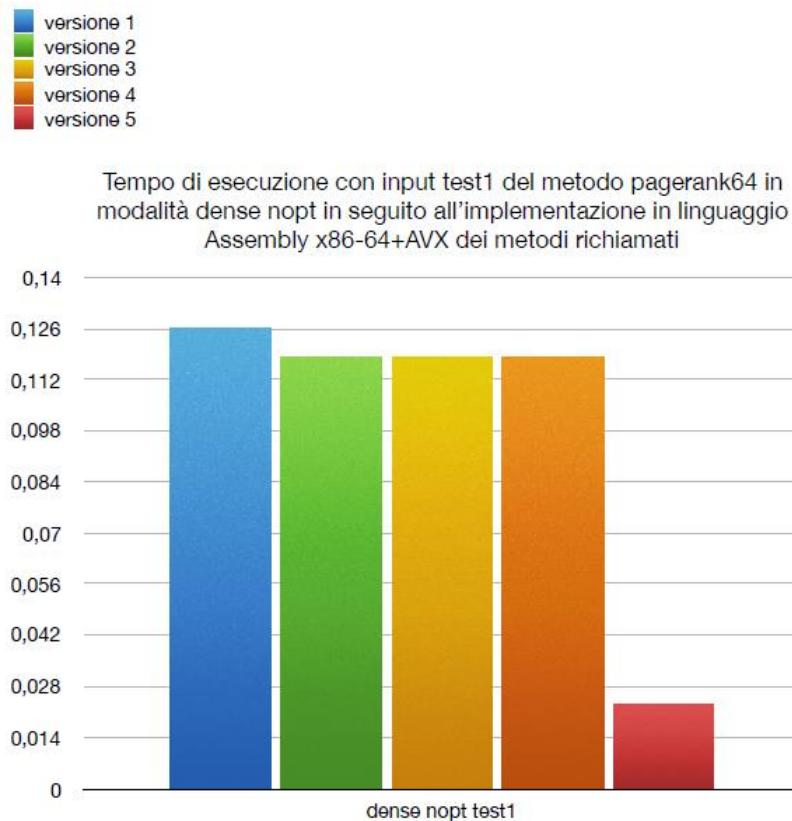
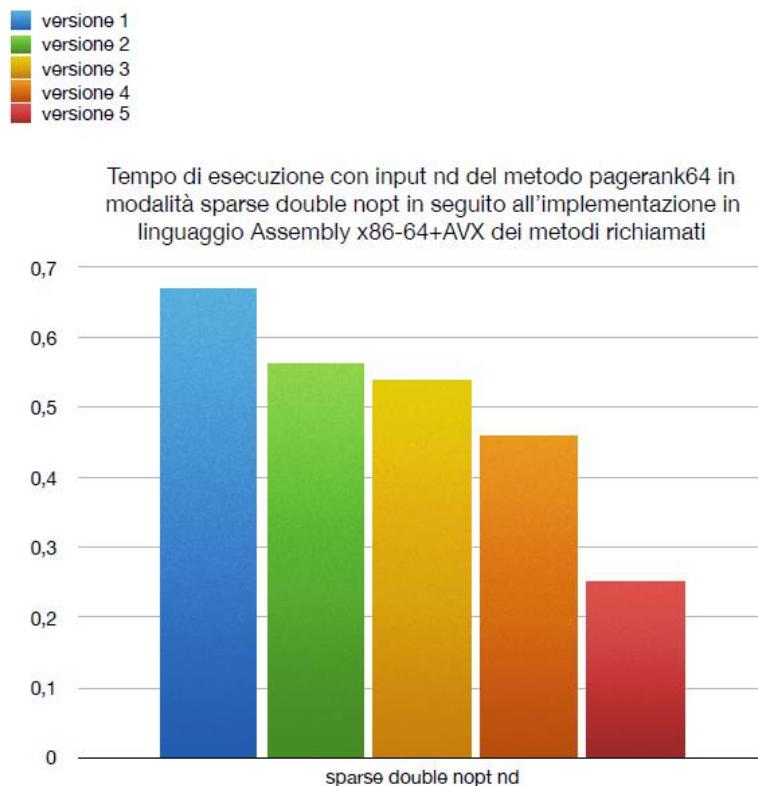


Di seguito sono riportati invece i grafici riguardanti le prestazioni a seguito delle traduzione in Assembly con l'architettura **x86-64+AVX**.



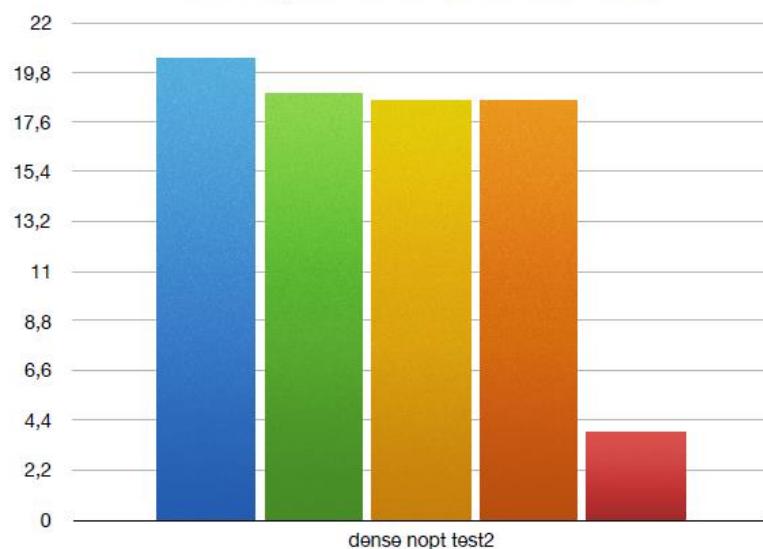




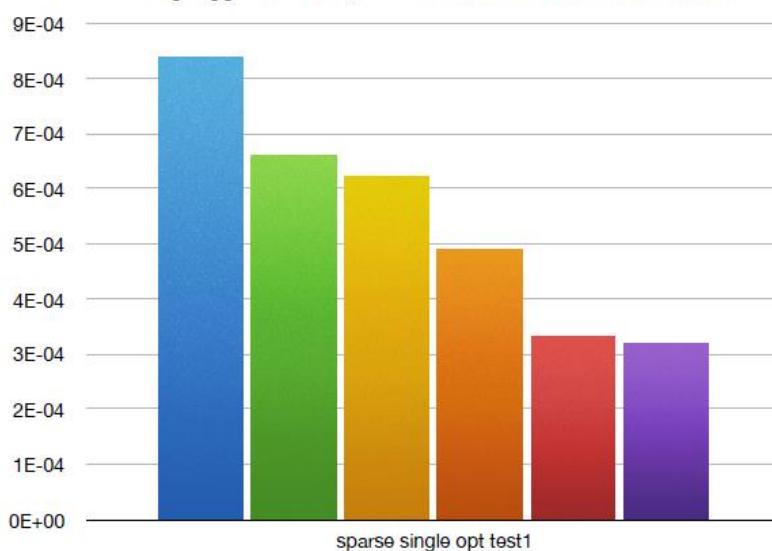


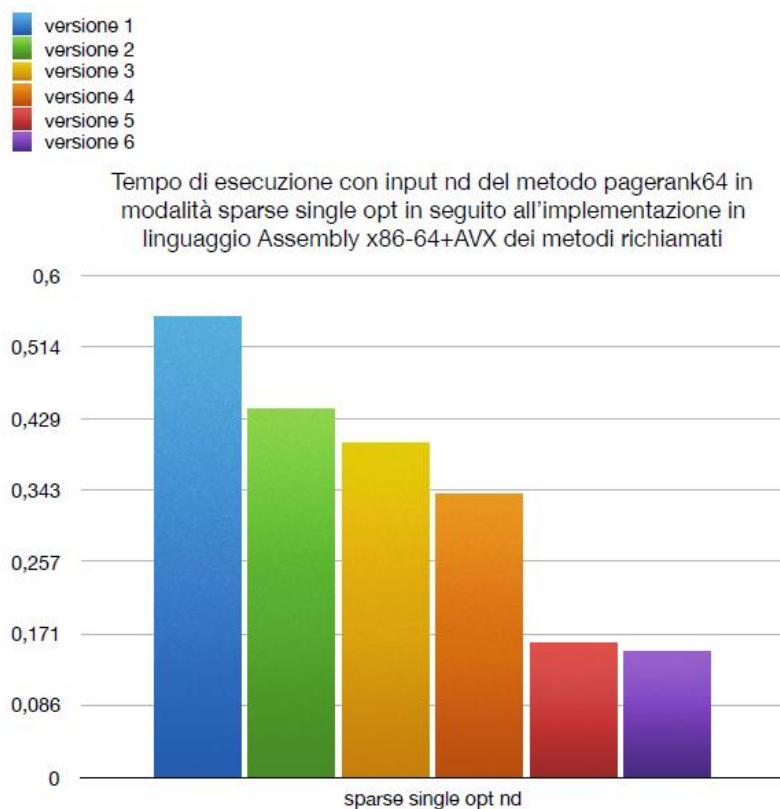
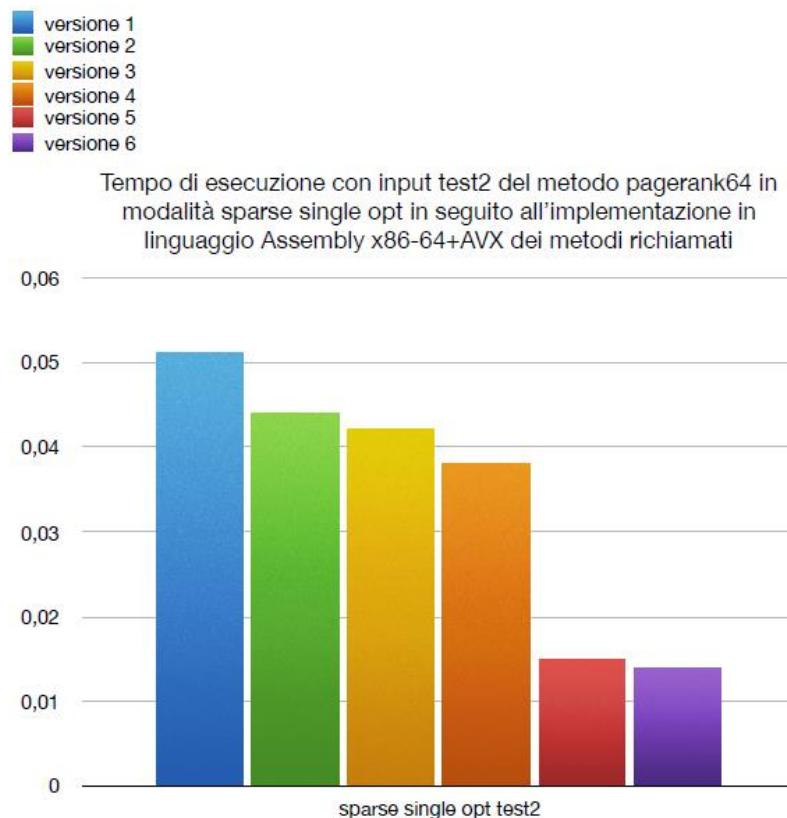


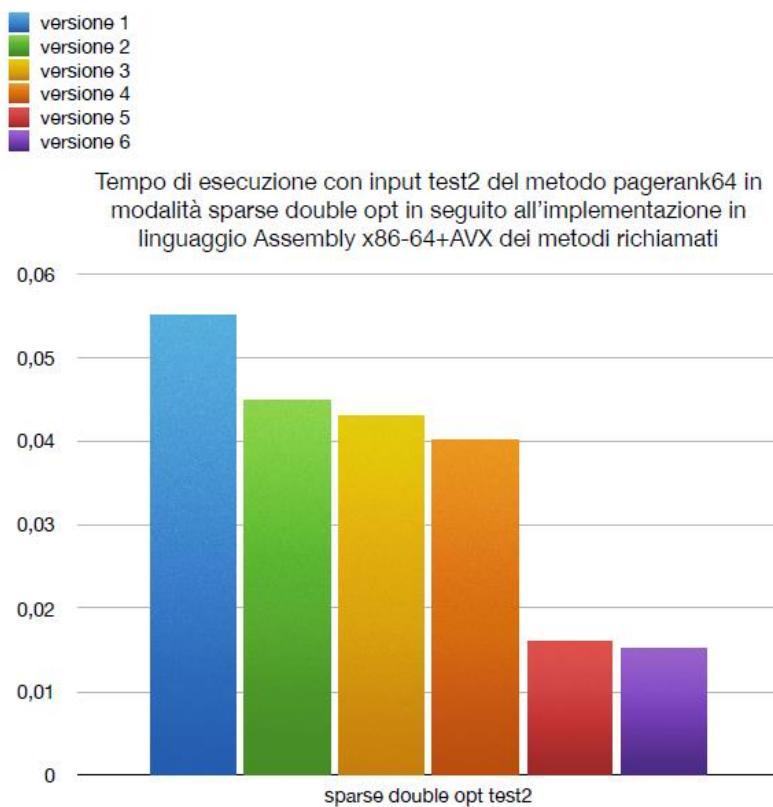
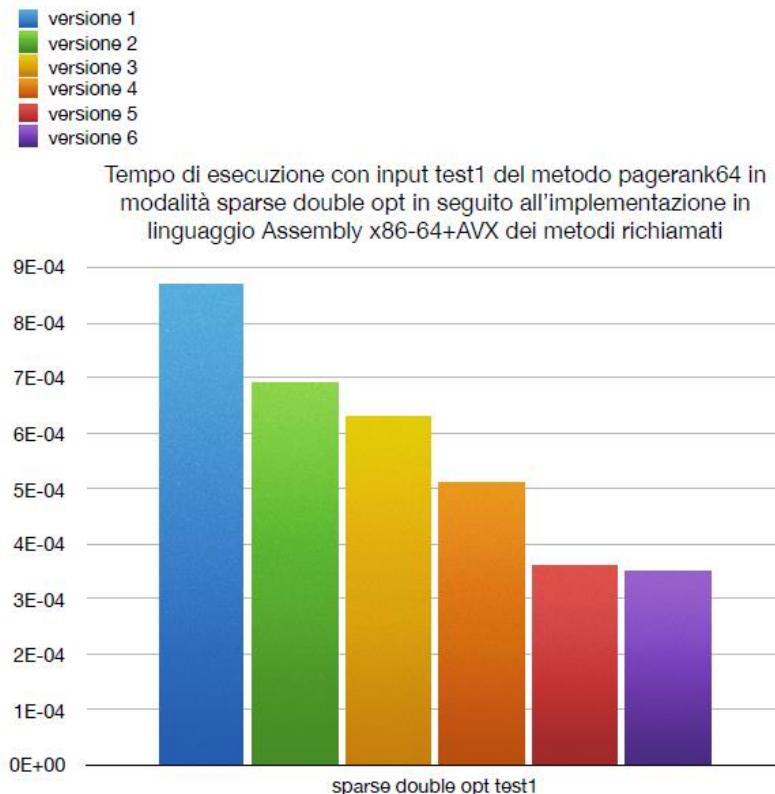
Tempo di esecuzione con input test2 del metodo pagerank64 in modalità dense nopt in seguito all'implementazione in linguaggio Assembly x86-64+AVX dei metodi richiamati

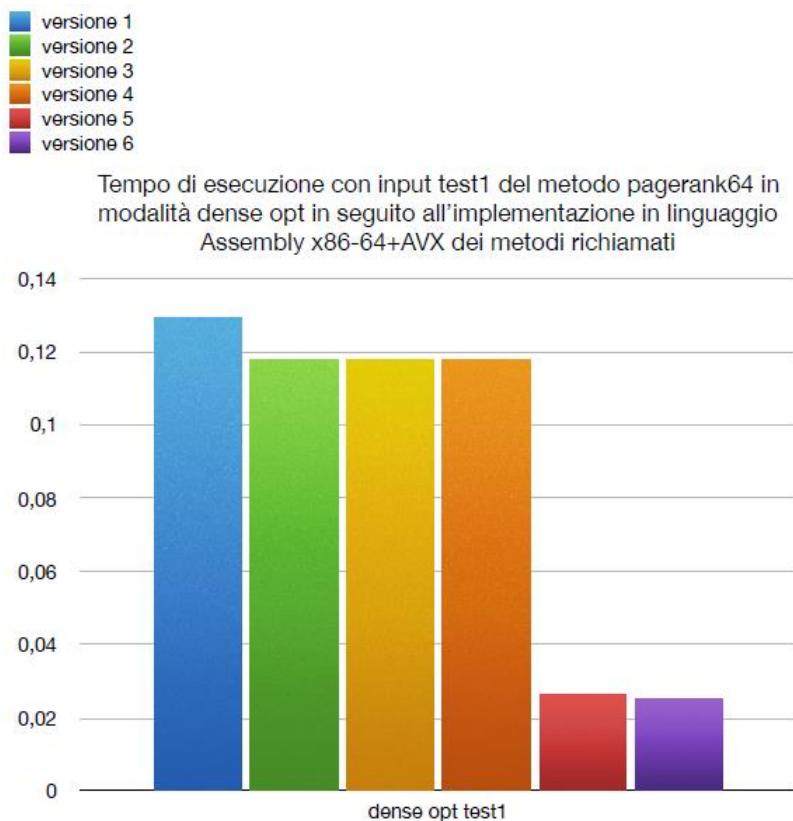
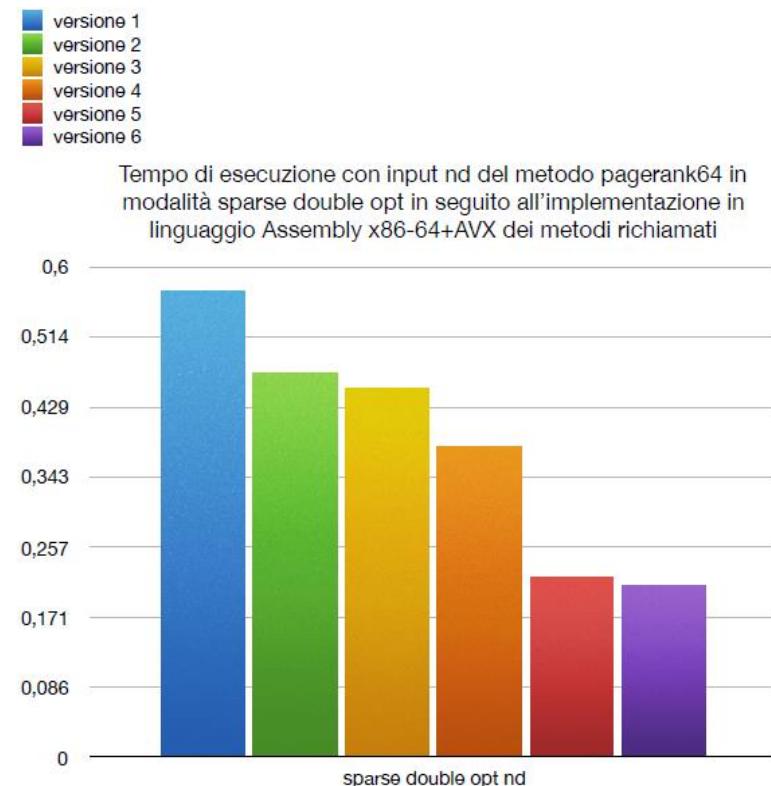


Tempo di esecuzione con input test1 del metodo pagerank64 in modalità sparse single opt in seguito all'implementazione in linguaggio Assembly x86-64+AVX dei metodi richiamati



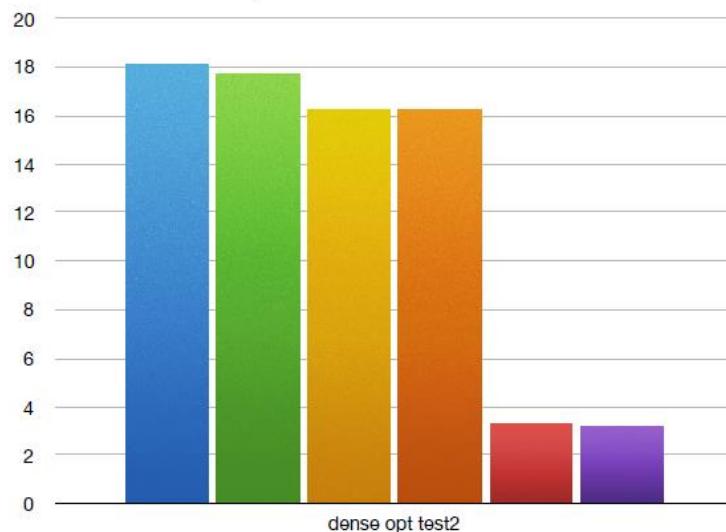






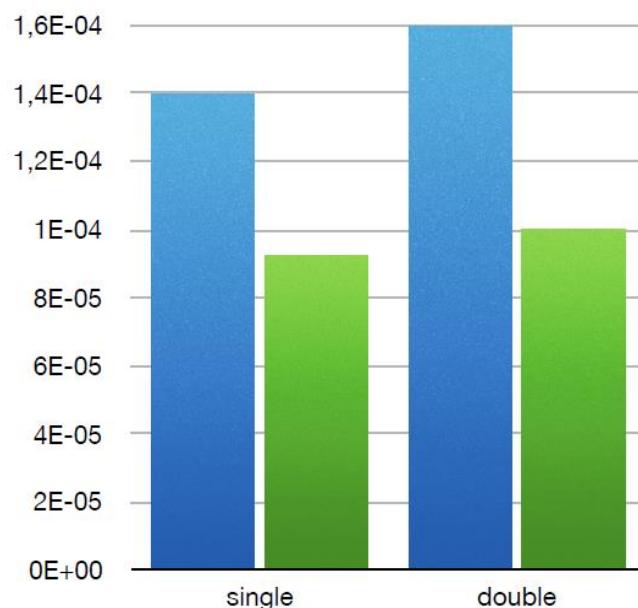
- █ versione 1
- █ versione 2
- █ versione 3
- █ versione 4
- █ versione 5
- █ versione 6

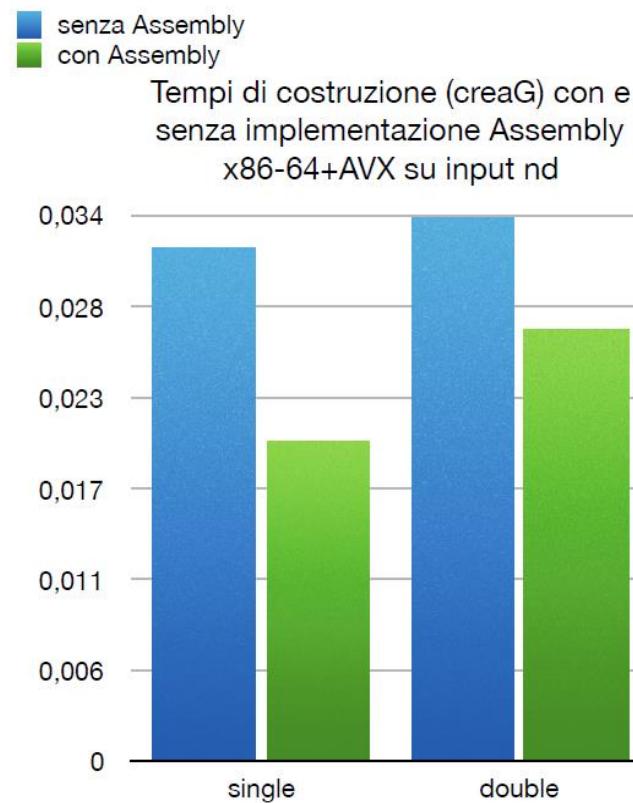
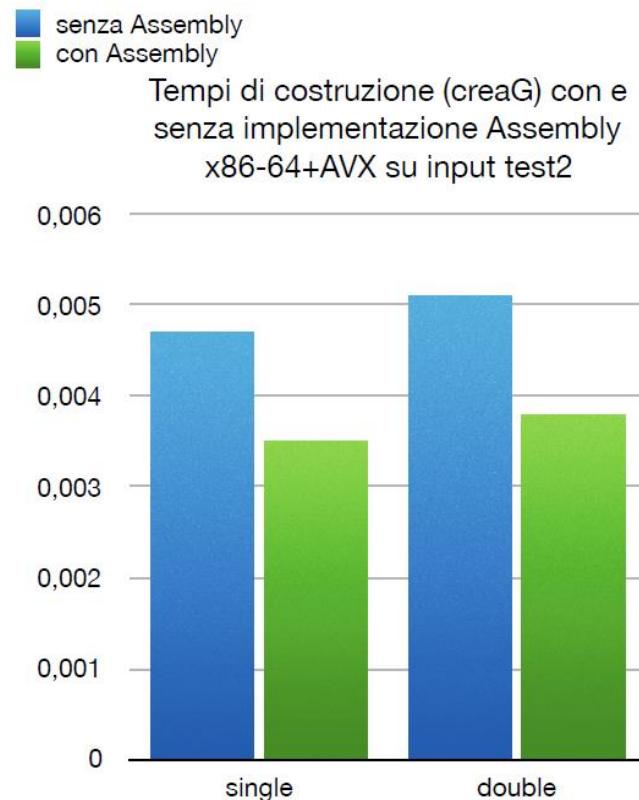
Tempo di esecuzione con input test2 del metodo pagerank64 in modalità dense opt in seguito all'implementazione in linguaggio Assembly x86-64+AVX dei metodi richiamati



- █ senza Assembly
- █ con Assembly

Tempi di costruzione (creaG) con e senza implementazione Assembly x86-64+AVX su input test1





Come si può notare dai grafici, tali versioni presentano lo stesso andamento della versione con l'architettura x86-32+SSE, ma si può notare un ulteriore miglioramento delle prestazioni dovuto al grado di parallelismo più elevato rispetto a prima.

Confronto tra la soluzione proposta e implementazioni alternative

Per verificare la bontà delle scelte progettuali effettuate sono stati fatti dei confronti con altre versioni che implementano l'algoritmo del PageRank in maniera diversa a come illustrata fino a questo momento.

In primo luogo si è dovuto verificare, per quanto riguarda la versione dense, che la rappresentazione *column-major order* sia effettivamente conveniente rispetto a quella *row-major order*. Come ci si aspettava la soluzione scelta risulta più conveniente come si evince dai tempi di esecuzione qui riportati degli algoritmi scritti in codice C.



Tale miglioramento deriva dal fatto che salvare la matrice in *column-major order* preserva la località spaziale in quanto nel prodotto implementato gli elementi sono acceduti seguendo un ordine sequenziale lungo le colonne.

Come per la versione dense, anche per quanto riguarda la versione sparse la scelta della rappresentazione dell'input è stata di fondamentale importanza per la massimizzazione delle prestazioni. In particolar modo sono stati esaminate altre due possibili strutture per rappresentare il grafo ovvero:

- Array di Location
- Compressed Sparse Row (CSR)

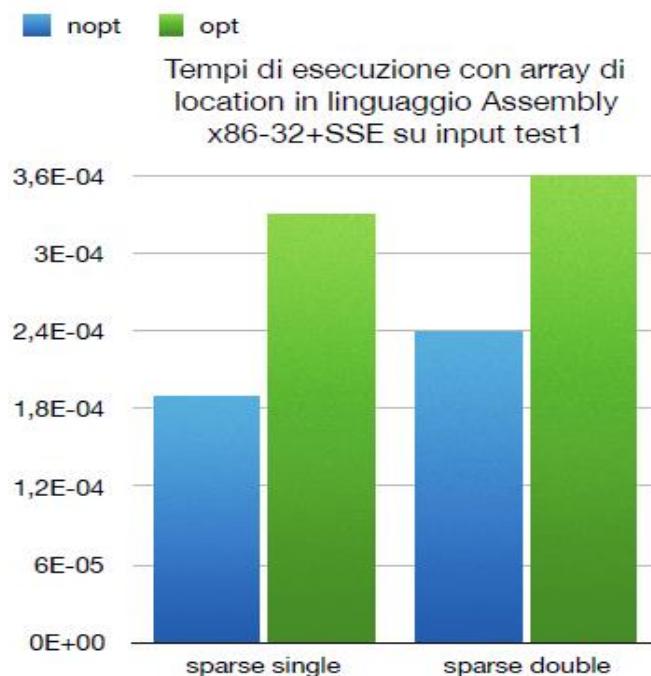
Come detto nelle sezioni precedenti, l'input dell'algoritmo nella versione sparse è salvato in un array di archi che sono rappresentati da uno struct detto *Location*. Un possibile approccio all'implementazione dell'algoritmo PageRank è dunque quello di non modificare in alcun modo tale struttura, ma utilizzare direttamente l'**array di location** per realizzare il prodotto sparso come si può vedere nel seguente frammento di codice.

```
void prodSparseF(GRAPH L, float* vf, float* xk1, float* xk, int m){  
    int i,j,sorg,dest;  
  
    for(i =0; i<m;i++){  
        sorg= L[i].x;  
        dest= L[i].y;  
        xk[dest] += vf[sorg]*xk1[sorg];  
    }  
}
```

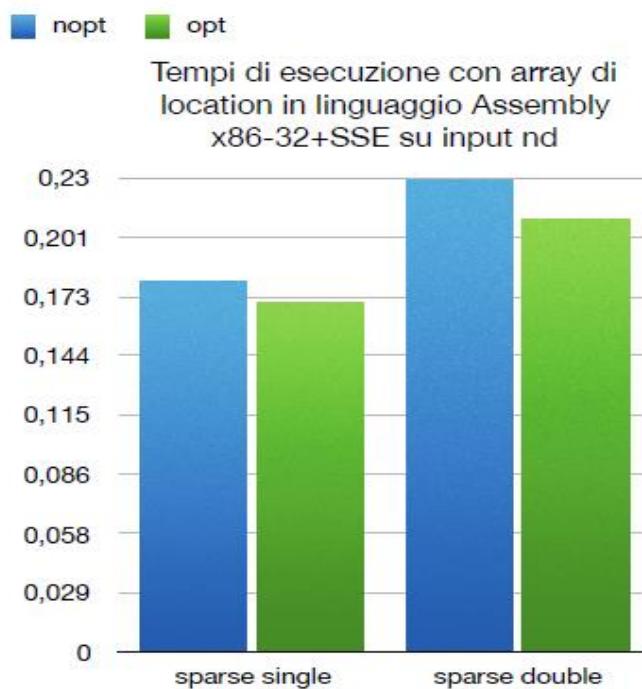
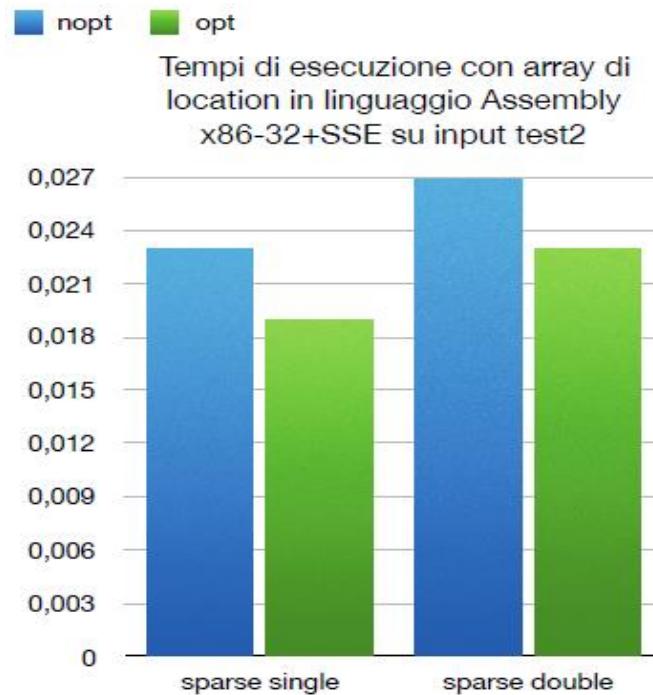
Il primo vantaggio evidente che si può notare è che non vi è più un ritardo dovuto alla manipolazione dell'input per ottenere una ulteriore struttura come quella del **CSC**. Tale vantaggio è evidente soprattutto quando le istanze sono di taglia minore, in quanto il tempo di costruzione assume un'importanza non banale rispetto al tempo di convergenza del metodo delle potenze. Tale struttura presenta però anche dei difetti che derivano soprattutto dalla casualità con la quale gli archi che compongono il grafo sono disposti nell'array. A causa di ciò infatti risulta impossibile precalcolare i prodotti parziali come mostrato nella soluzione adottata, ma sarà invece necessario calcolarli nnz volte anziché n . Inoltre, ad ogni calcolo di un prodotto parziale, è necessario effettuare un accesso in memoria per aggiornare il valore di una pagina, per cui è necessario effettuarne nnz anziché

n come nella soluzione implementata. Tali svantaggi portano dunque a delle prestazioni peggiori nel calcolo del prodotto rispetto alla struttura del CSC, per cui per input di taglia più grande dove il tempo di costruzione diventa trascurabile rispetto a quello del prodotto la scelta della struttura del CSC risulta la migliore.

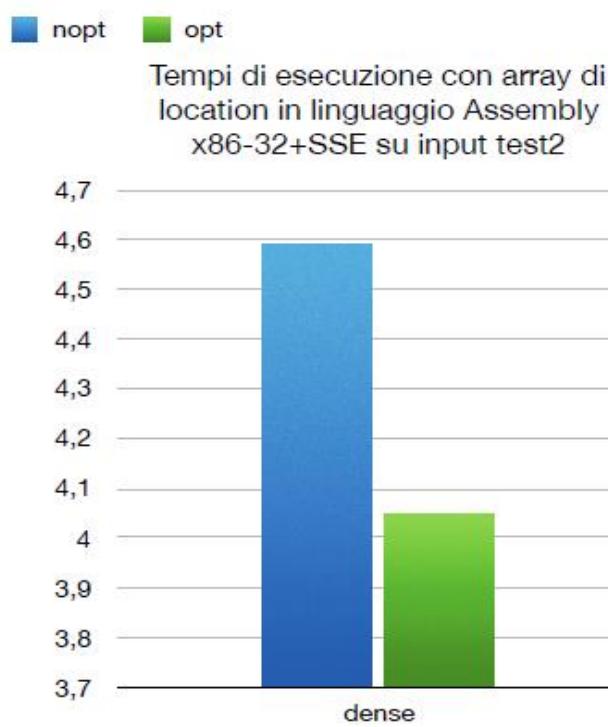
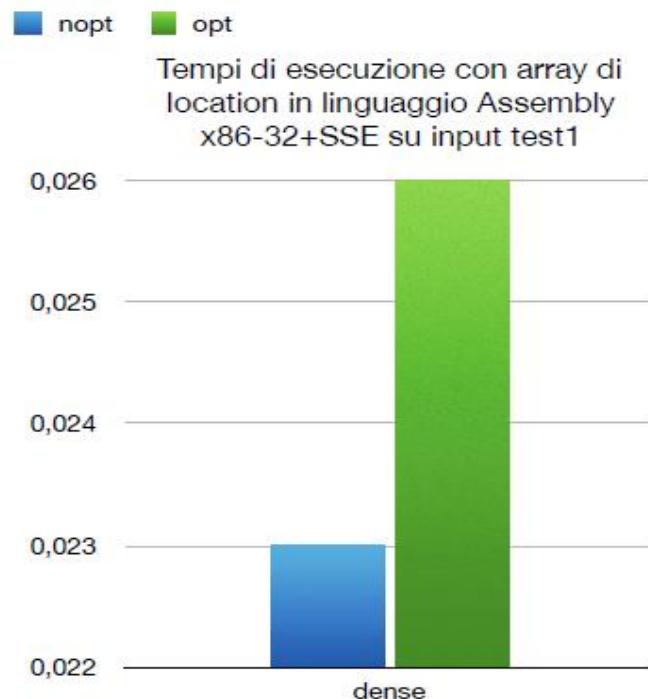
Anche l'algoritmo che utilizza l'**array di location** è stato tradotto interamente in linguaggio Assembly in relazione all'architettura x86-32+SSE, e i grafici qui riportati mostrano i tempi di convergenza raggiunti con i 3 input *test1*, *test2* ed *nd*.



Confronto tra la soluzione proposta e implementazioni alternative 50



Confronto tra la soluzione proposta e implementazioni alternative 51



Come ci si aspettava, tale algoritmo ha dei tempi migliori rispetto all'algoritmo utilizzato nel progetto con *test1* in quanto è un'istanza di piccole dimensioni, mentre con istanze più grandi come *test2* ed *nd* la soluzione scelta si rivela la migliore.

Una soluzione alternativa per la rappresentazione del grafo è il **CSR**, una struttura simile al CSC in quanto composta anch'essa da 3 array:

- **values**, di lunghezza nnz , che contiene i valori non nulli della matrice memorizzati da sinistra a destra e da sopra verso sotto.
- **column**, di lunghezza nnz , che contiene gli indici di colonna corrispondenti ai valori non nulli della matrice.
- **rowPtr**, di lunghezza $n + 1$, che contiene nella posizione i -esima l'indice del primo elemento di *values* diverso da 0 della riga i -esima. La lunghezza della riga i è determinata da $rowPtr[i+1]-rowPtr[i]$.

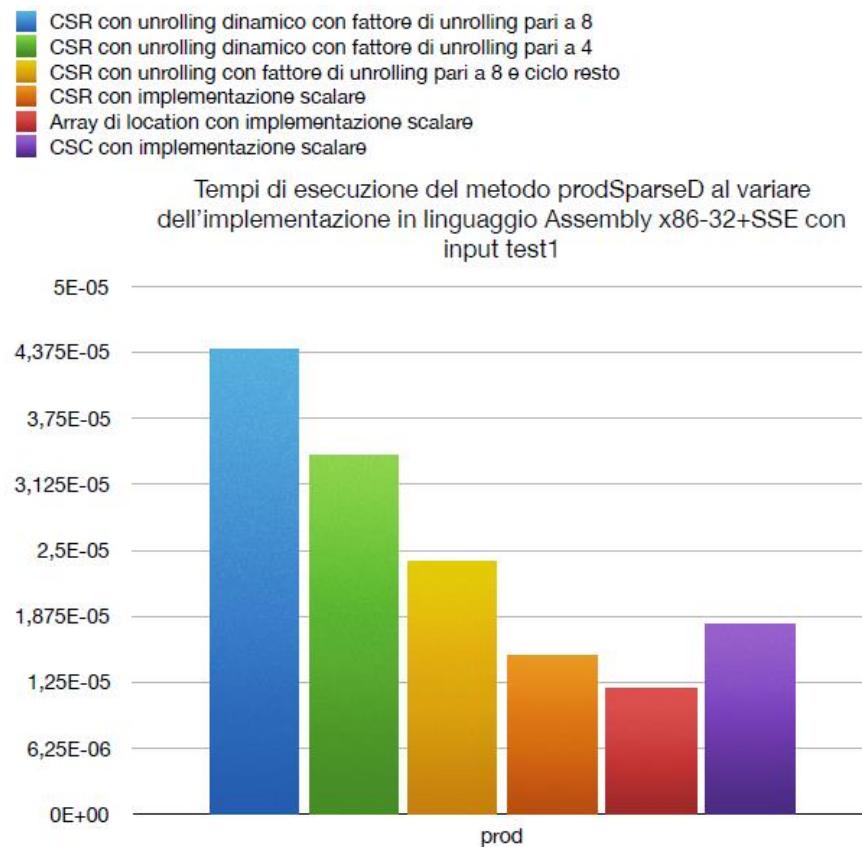
Come nel caso del CSC, è necessario trasformare l'input, inizialmente sotto forma di array di location, per creare tale struttura prima di eseguire l'algoritmo e pertanto anche in questo caso si ha un'overhead a seguito della costruzione.

L'implementazione del prodotto utilizzando tale struttura può essere vista nel seguente frammento di codice

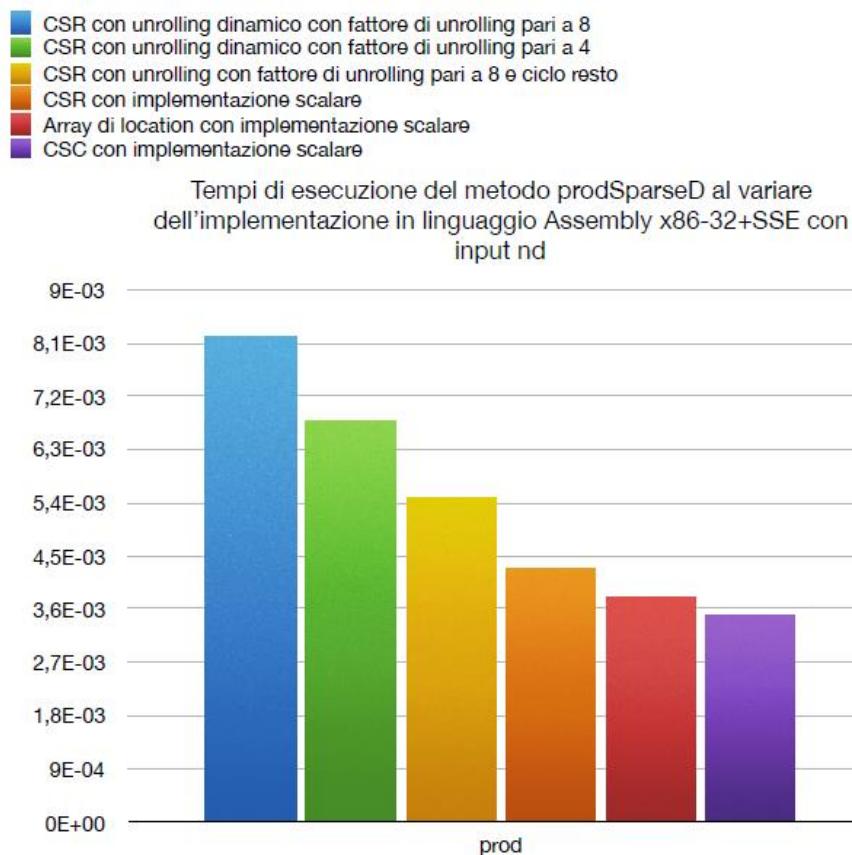
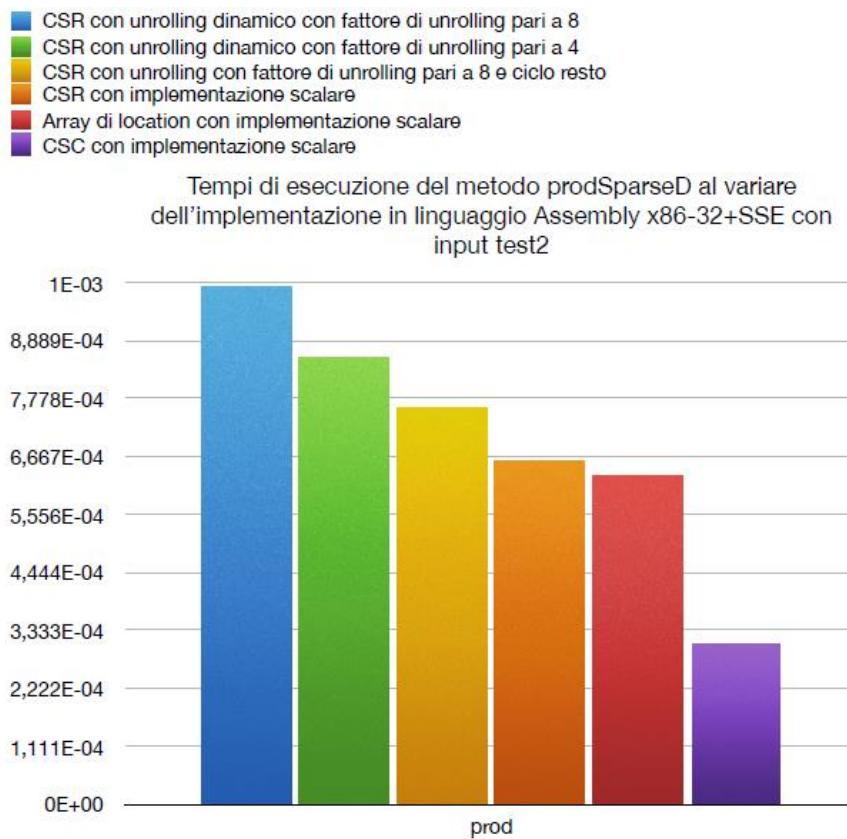
```
void prodSparseD(int* row, int* col, VECTORD vd, int N, VECTORD xk1, VECTORD xk){
    int i,j, rowel, curcol=0;
    double s;
    for(i=0;i<N;i++){
        rowel = row[i];
        for(j=0; j<rowel; j++){
            xk[col[curcol]] += vd[curcol]* xk1[i];
            curcol++;
        }
    }
}
```

Il difetto di tale struttura deriva dal fatto che, per effettuare il prodotto matrice-vettore, è necessario scorrere la matrice per colonne e non per righe. Così facendo si devono quindi effettuare nnz accessi indiretti in memoria per aggiornare i valori parziali di PageRank associati a ciascuna pagina al calcolo di ciascun prodotto parziale, anziché effettuarne solo n sequenziali come nel caso del CSC.

Di seguito sono riportati i tempi medi necessari per effettuare il prodotto utilizzando le strutture finora descritte.



Confronto tra la soluzione proposta e implementazioni alternative 54



Come si evince dalle figure, la soluzione implementata con il CSC presenta il prodotto più efficiente per istanze più grandi e, per tale motivo, è stata preferita alle altre strutture. Un altro aspetto da notare è che sono stati presi in considerazione più versioni del prodotto con il CSR, i quali differiscono tra loro per l'implementazione di diversi tipi di ciclo resto. Come detto in precedenza infatti, non risulta conveniente effettuare il padding per realizzare il prodotto sparso a causa dell'elevato numeri di 0 introdotti nella matrice. Pertanto, per parallelizzare le istruzioni, è stato analizzati diversi approcci basati sul ciclo resto:

- Ciclo quoziente di 8 e ciclo resto di 4,2 e 1
- Ciclo quoziente di 8 e ciclo resto di 2 e 1
- Ciclo quoziente di 8 e ciclo resto di 1

Quello che si evidenzia dai tempi mostrati dal grafico in figura è che la soluzione scalare con il CSR è risultata la migliore tra tutte quelle relative alla stessa struttura in quanto i cicli quoziente e resto introducono un overhead a causa degli eccessivi controlli e dunque di salti condizionati. A seguito di tali risultati si è scelto perciò di implementare il CSC solamente utilizzando un ciclo scalare.