Relazione MinHash

Algoritmo di minhash sviluppato in versione seriale e parallela, utilizzando Pthread e OpenMP

1. Presentazione……...pag. 2-4

1.1 Minhash
1.2 Stima della similarità
1.3 Clustering di documenti
1.4 Filtering Near-Duplicates
1.5 Approccio scelto in questo progetto

2. Algoritmo……..……...pag. 5-14

1.1 Implementazione algoritmo seriale
1.2 OpenMP
1.3 Pthreads
1.4 Produttore-Consumatore

3. Test…….pag.

1. Presentazione

1.1 MinHash

MinHash è una tecnica algoritmica per stimare velocemente il grado di somiglianza di due insiemi. È stato inizialmente utilizzato dal motore di ricerca AltaVista per identificare duplicati tra le pagine web ed eliminarli dai risultati di ricerca. Viene anche utilizzato per raggruppare documenti per somiglianza dei loro insiemi di parole [1].

Per il calcolo della somiglianza tra insiemi si usa comunemente il coefficiente di similarità di Jaccard, che è definito dal rapporto tra la dimensione dell'intersezione e la dimensione dell'unione degli insiemi campionari [2] :

$$J(A,B) = \frac{|A \cap B|}{|A \cup B|}. \tag{1}$$

L'obbiettivo del MinHash è di stimare velocemente J(*A,B*) senza calcolare esplicitamente l'intersezione e l'unione, questo per poter lavorare su grandi quantità di dati rendendo i tempi accettabili. Tramite una funzione di hash e delle permutazioni applicate a insiemi di dati, di cui vengono ricavati dei valori di hash minimi, il minHash produce una stima (probabilistica) approssimativa dell'Indice di Jaccard.

Ogni documento, dopo essere stato tokenizzato (eliminando formattazione, maiuscole, ecc) viene ridotto in tempo lineare a uno *sketch* di qualche centinaia di bytes composto da una collezione di *fingerprint* (o *signature*, ottenute tramite una funzione di hash e permutazioni) di *shingle*. Quest'ultimi sono sottosequenze contigue di parole di lunghezza fissa che differiscono l'una dall'altra per un solo elemento e vengono usati per rendere i documenti compatibili con il problema dell'intersezione. Per convenienza non vengono usati direttamente, ma mappati in *signatures* da 64 bit, con il presupposto che se due *signatures* sono uguali, con grande probabilità anche gli oggetti a cui si riferiscono lo sono. Per ridurre al minimo l'impatto delle collisioni sul risultato finale si applica una permutazione randomica sugli *shingle* e il *fingerprint* del valore minimo (in pratica sono sufficienti permutazioni pseudo-randomiche) [3].

1.2 Stima della similarità

Poniamo di avere due documenti *A* e *B*, e per ciascuno un insieme S_A e S_B delle *signatures* di 64 bit ottenute dai rispettivi *shingles*. *S* sarà quindi un sottoinsieme di S_n, l'insieme cioè dei valori in $[2^{64}]$. Sia π una permutazione presa randomicamente dall'insieme di tutte le possibili permutazioni di $[2^{64}]$[1]: la probabilità che il minimo della permutazione π sui valori in S_A sia uguale al minimo della permutazione π sui valori in S_B è uguale all'Indice di Jaccard su *A* e *B*:

$$\Pr\big(\min\{\pi(S_A)\} = \min\{\pi(S_B)\}\big) = \frac{|S_A \cap S_B|}{|S_A \cup S_B|} \tag{2}$$

1. Dato un insieme *X*, una permutazione è un'applicazione biiettiva da *X* in *X*, quindi π mapperà i valori in S_A e S_B in valori in $[2^{64}]$. [4]

Per dimostrare questa uguaglianza, notiamo che applicando una permutazione π su X, tutti gli elementi x in X hanno uguale probabilità di diventare il minimo *(*detta *min-wise independence condition)*:

$$\Pr\big(\min\{\pi(X)\} = \pi(x)\big) = \frac{1}{|X|}. \tag{3}$$

Sia α la più piccola immagine in $\pi(S_A \cup S_B)$: i minimi valori della permutazione π su S_A e S_B sono uguali se e solo se α è immagine di un elemento in $(S_A \cap S_B)$. Quindi:

$$\begin{aligned}\Pr\big(\min\{\pi(S_A)\} = \min\{\pi(S_B)\}\big) &= \Pr(\pi^{-1}(\alpha) \in S_A \cap S_B) \\ &= \frac{|S_A \cap S_B|}{|S_A \cup S_B|} = r_w(A,B).\end{aligned} \tag{4}$$

Per stabilire la somiglianza tra *A* e *B* sarà sufficiente stabilire in anticipo un numero *t* di permutazioni in modo tale che S_A e S_B siano gli insiemi dei minimi di queste permutazioni. Così per ogni documento, indipendentemente dal numero di *shingles*, si ricava uno stesso numero di *signatures* ottenute tramite questa operazione di *min-Hash*. In pratica, poiché è impossibile avere permutazioni randomiche distribuite uniformemente su S_n è possibile scegliere un piccolo sottoinsieme di queste permutazioni che soddisfino la *min-wise independence condition* (3), poiché questa è condizione sufficiente per la (2) [3], che verranno applicate a tutti i documenti da confrontare.

La permutazioni applicate agli *shingles* fanno sì che eventuali collisioni abbiano un impatto modesto sul risultato, permettendo quindi di avere *signatures* di dimensioni relativamente piccole (64 bit). Uno *sketch* sarà quindi precisamente la collezione ordinata di *signatures* di un documento ottenuta da questa operazione di *min-Hashing* degli *shingles*.

1.3 Clustering di documenti

L'utilizzo degli *sketches* permette di raggruppare una collezione di *m* documenti in un insieme di documenti simili in tempo O(mlogm) invece che O(m^2). Una volta calcolati gli *sketches* nel modo sopra illustrato, il *clustering* avviene in ulteriori 3 fasi:

1. Tutti gli *sketches* di tutti i documenti vengono espansi in una lista di coppie {signature, ID documento}, ordinata per il valore delle *signatures.*

2. Viene generata una lista di tutte le coppie di documenti che condividono almeno una *signature*, insieme al numero di *signatures* che hanno in comune. Per fare questo si genera una lista di triple {ID doc1, ID doc2, 1} per ogni *signature* condivisa dai documenti. Si ordina questa lista per ID doc1 e si salva la somma del numero delle *n* signatures condivise in {ID doc1, ID doc2, *n*}.

3. Si esamina ogni tripla {ID doc1, ID doc2, *n*} e si valuta se supera la soglia scelta di similarità.

1.4 Filtering Near-Duplicates

Per ridurre ulteriormente la dimensione delle *signatures* e il costo computazionale, si usa la tecnica dei *Near-Duplicates*. Due documenti quasi identici condivideranno quasi tutte le *signatures*, che possono essere raggruppate in *k* gruppi di *s* elementi ciascuno, con *k* e *s* opportunamente scelti [3]. Ogni gruppo viene ridotto in una nuova *fingerprint*, anche questa di 64 bit, denominata *feature*. Gli sketches di ogni documento consisteranno quindi di *k features*, riducendo di *s* volte lo spazio richiesto. Queste *features* vengono poi confrontate tra coppie di documenti, in base all'indice i_k del gruppo di appartenenza. In generale i documenti che condividono almeno 2 features possono essere considerati duplicati e uno dei due può essere eliminato.

1.5 Approccio scelto in questo progetto

Il calcolo della somiglianza tramite *clustering*, che confronta le *signatures* senza tenere conto del loro ordine rispetto al documento da cui sono state generate, permette di individuare parti comuni nei documenti anche se si trovano in posizione diverse. È dunque possibile ottenere un indice di similarità molto preciso, ma non è possibile determinare l'identicità dei documenti per quanto riguarda l'ordine delle parti. La tecnica dei *Near-Duplicates* permette invece, confrontando ordinatamente le *features (*che sono una rappresentazione ridotta delle *signatures)*, di individuare coppie di documenti che hanno una probabilità molto alta di essere identici. Questa seconda tecnica, non aggiungendo nulla rispetto alla prima in termini di complessità algoritmica e parallelizzazione, non è stata implementata in questo progetto.

2. Implementazione algoritmo

2.1 Implementazione algoritmo seriale

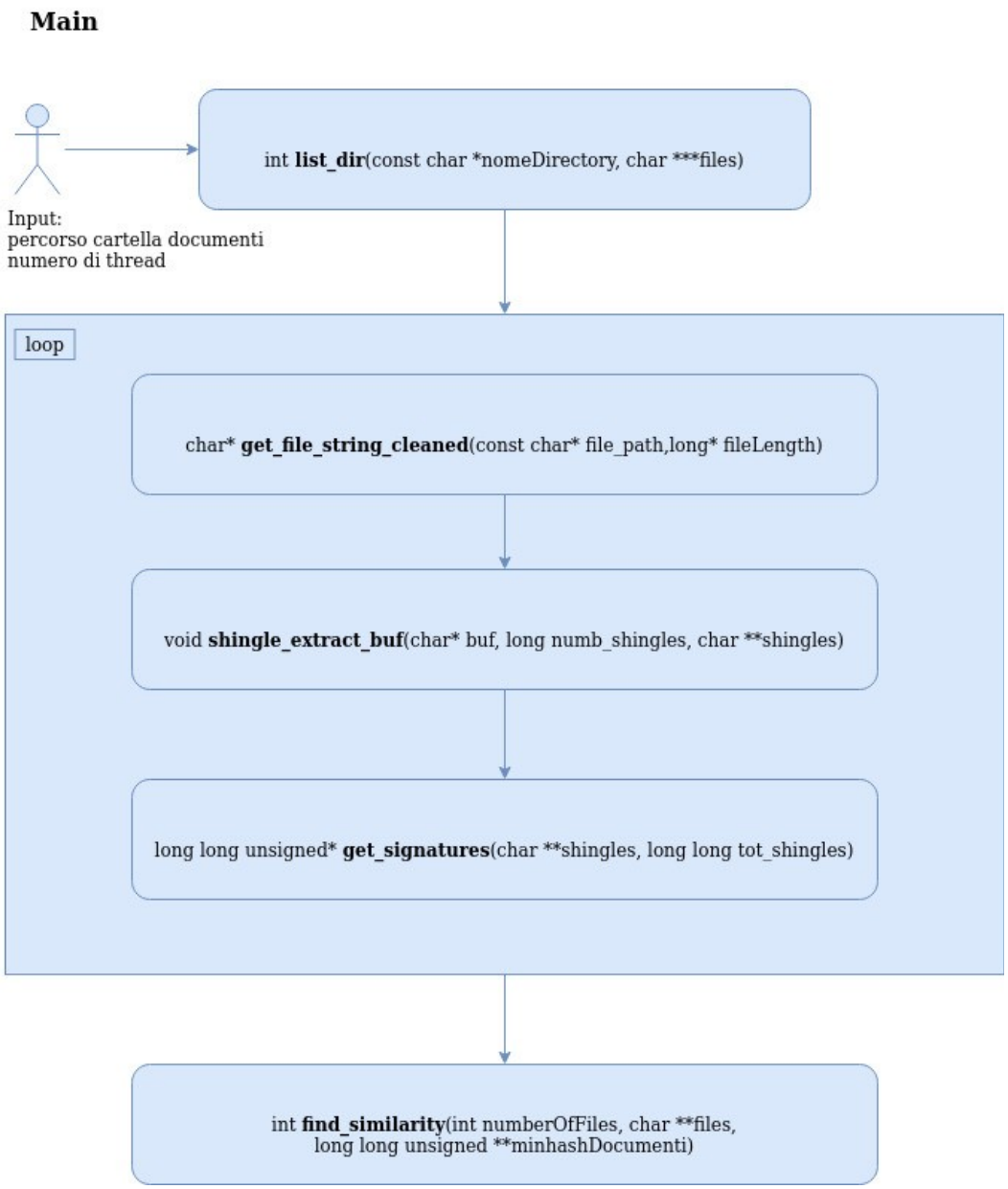
Il programma è composto da più moduli. Il principale è il file main.c che richiama a sua volta i file: documents_getters.c, tokenizer.c, shingle_extract.c, hash_FNV_1.c, get_similarities.c e per fini di test il file time_test.c .

Nel main.c viene avviata l'applicazione che prende come argomento il percorso dei documenti da verificare, assumendo che siano file di testo. Il recupero dei nomi dei documenti viene fatto con la funzione list_dir che preso in input il path della cartella va a popolare per riferimento un array di stringhe con i nomi dei files e restituisce la grandezza dell'array.

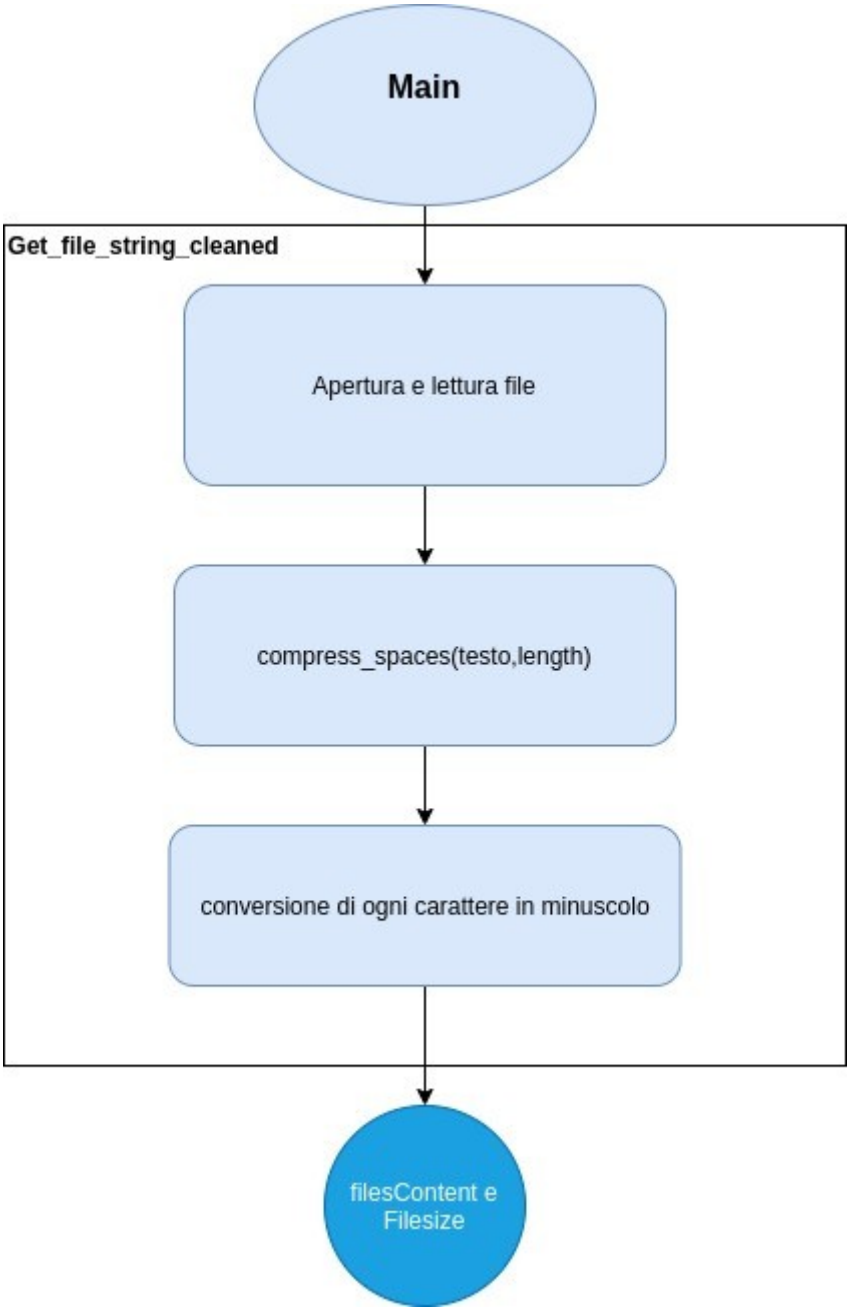
Ogni file trovato viene elaborato con le seguenti funzioni:

- il **get_file_string_cleaned** che si occupa di recuperare il contenuto del file e ripulirlo dagli spazi, formattatura, maiuscole e caratteri non ASCII.
- **shingle_extract_buf** estrae gli *shingles* dal testo tokenizzato e li copia in una array di stringhe.
- **get_signatures** crea le *signatures* applicando la funzione di hash e 199 permutazioni.

Raccolte tutte le *signatures* per tutti i files, la funzione **find_similarity** si occupa dell'ordinamento e della comparazione delle *signatures* e calcola la similarità tra i documenti.



Nel file **tokenizer.c** dati i percorsi dei files vengono ricavati e ripuliti da caratteri non conformi i contenuti testuali. Inizialmente viene solo ricavato l'intero testo e salvato in un puntatore, poi vengono rimossi spazi,caratteri non ASCII e tab. L'ultima procedura prevede la conversione di eventuali caratteri maiuscoli in minuscoli a questo punto le stringhe sono pronte ad essere trasformate in *shingles*.

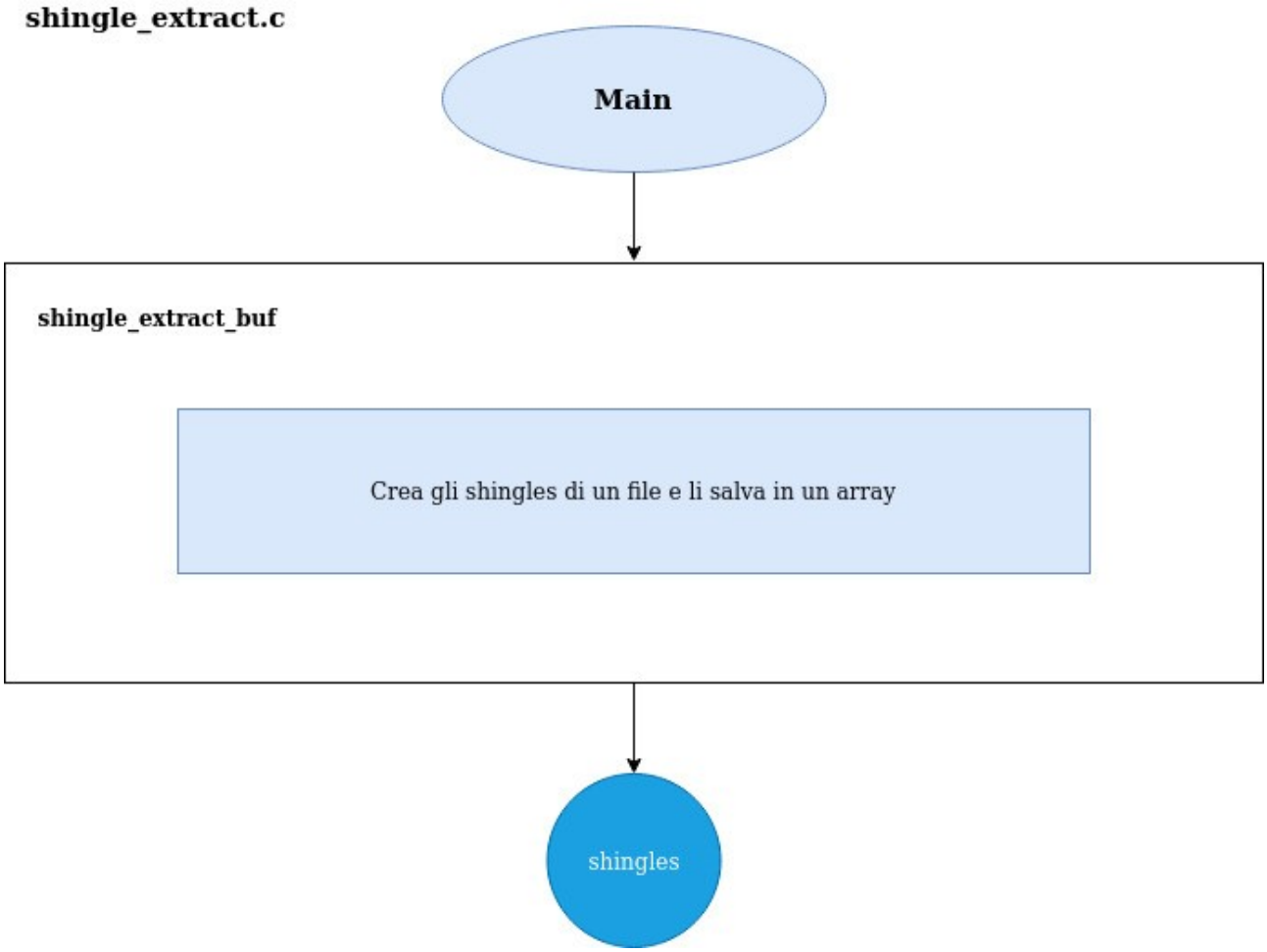


La costante K-SHINGLE permette di scegliere la lunghezza in caratteri degli *shingles*, che dovrà essere uguale per tutti i files da confrontare. Per questo programma è stato scelto il valore 54, che corrisponde a 9 parole da 6 caratteri l'una.

Nel **main**, dopo aver ottenuto la dimensione del file tokenizzato, viene calcolato il numero *N* degli *shingles* che dovranno essere creati per quel file, tramite la formula:

$$N = fileSize - K\_SHINGLE + 1$$

Poi viene chiamata la funzione **shingle_extract_buf** che estrae gli N *shingles* dalla stringa buf in cui è stato salvato il contenuto tokenizzato del file, e li salva tramite puntatore in una matrice di caratteri di dimensione (N x K_SHINGLE).



La funzione **get_signatures** prende il puntatore alla matrice di *shingles* e opera i seguenti passaggi:

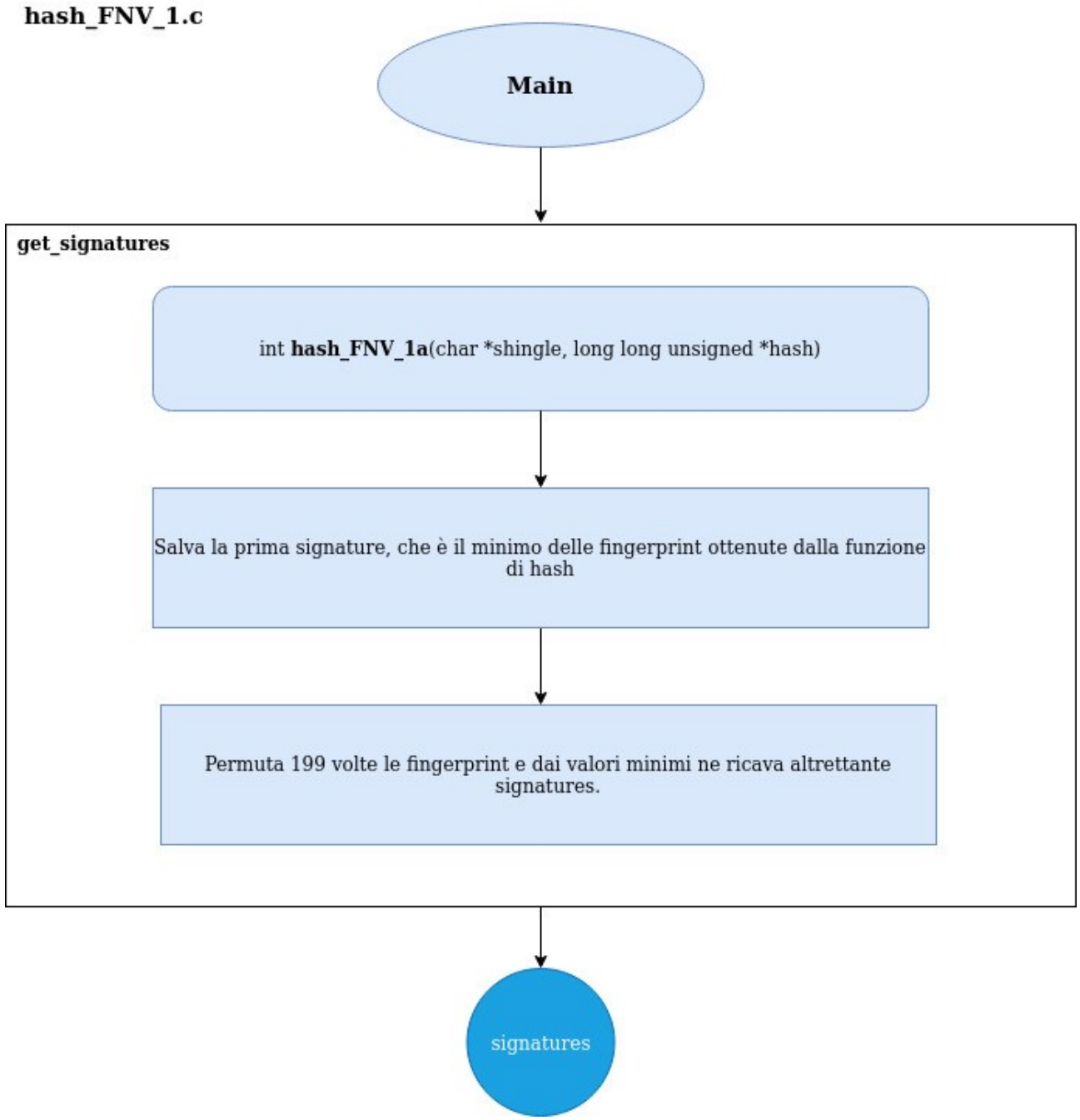
- converte ogni *shingle* in una *fingerprint* da 64 bit tramite la funzione **hash_FNV_1a**. Questa utilizza il numero primo *FNV_PRIME* (scelto per garantire buone proprietà di dispersione[5]) che moltiplica, per ogni carattere *i* dello *shingle*, il valore ottenuto al ciclo precedente (partendo da una base fissa da 64bit detta *FNV offset basis*) per poi applicargli a sua volta lo *XOR* con il carattere *i*.

- Tutte queste *fingerprints* che sono in numero pari al numero degli *shingles*, vengono temporaneamente salvate in un array. La minore di queste costituisce la prima *signature* per questo file.

- Con tutte le *fingerprints* viene poi eseguita l'operazione di XOR con 199 numeri random, che saranno sempre gli stessi per tutti i file, che servono a produrre le permutazioni pseudo-random necessarie per la (2).

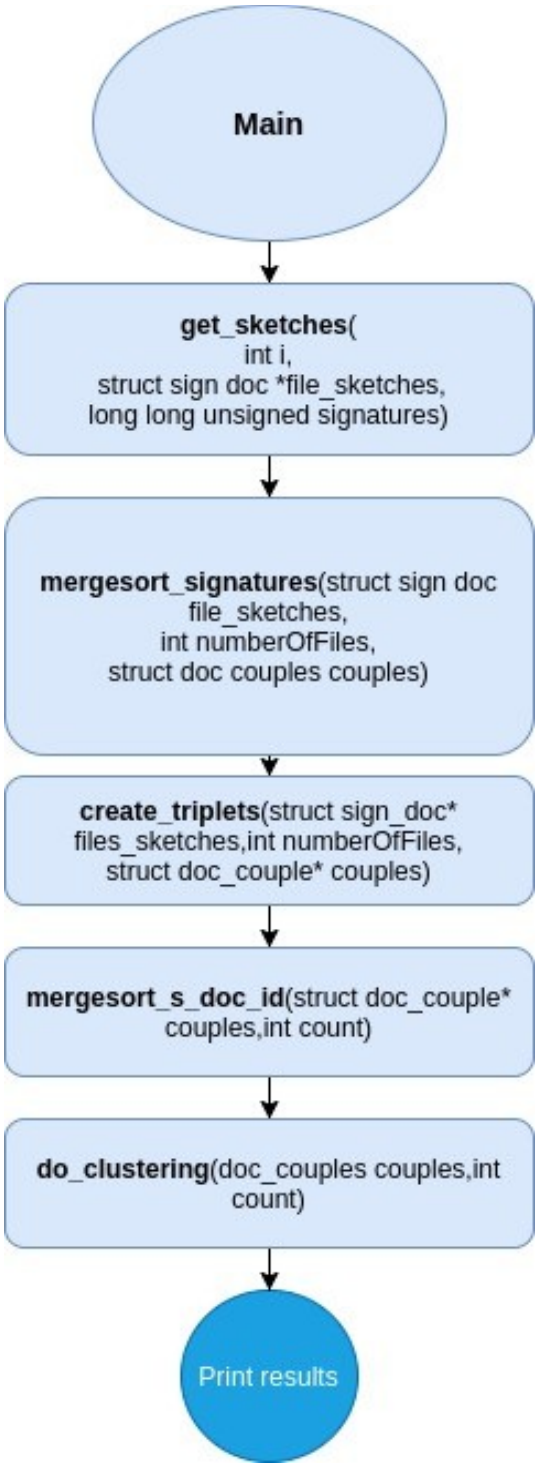
- Dopo ogni permutazione di tutte le *fingerprints*, si salva quella di valore minimo.

- Alla fine sono state create le 200 *signatures*, che vengono salvate nella matrice minhashDocumenti contenente tutte le *signatures* di tutti i documenti da confrontare.



Il **get_similarities.c** è l'ultimo modulo e si occupa di confrontare le *signatures* dei documenti e stampare eventuali similarità. All'inizio della funzione **find_similarity** si crea un array **sign_doc** una struttura dati contenente una *signature* del documento e il suo identificativo, subito dopo viene riordinata sulla grandezza delle *signatures* tramite l'algoritmo di mergesort.

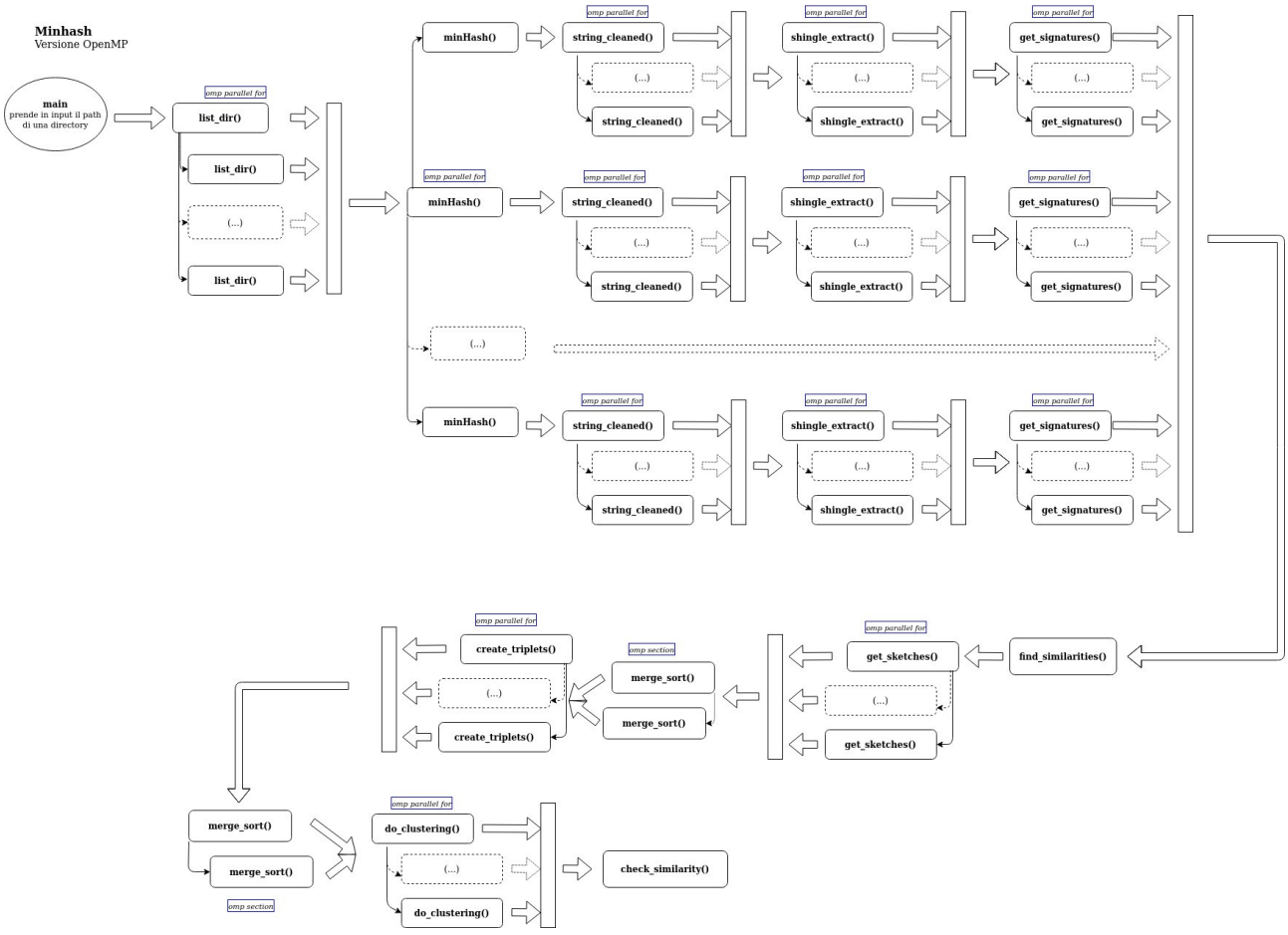
Una volta ordinati gli *sketches* con costo computazionale O(nlgn) vengono create delle couples sfruttando l'ordinamento: un array di struct **doc_couple** contenente i due doc_id con la *signature* in comune e il valore 1 (numero della signature).
L'obiettivo del restante codice è raccogliere il numero di shared signature in un unica couples mantenendo il costo computazionale di O(nlgn), per farlo si esegue un ordinamento delle couples di nuovo tramite mergesort sull'identificativo del primo documento. Di nuovo sfruttando l'ordinamento acquisito si va nella funzione **do_clustering** che tramite un primo loop su tutte le couples e uno annidato sulle successive couples va a sommare le shared signatures tra due documenti in un unica couples.
In tutti questi passaggi i relativi array di strutture dati una volta ridimensionati reallocano la memoria in modo da ridurre la memoria usata al minimo.



L’algoritmo si conclude con la stampa tramite la funzione **check_and_print_similarity** delle couples che contengono gli identificativi dei documenti, il numero di *signatures* in comune e il grado di similarità tra 0 e 1.

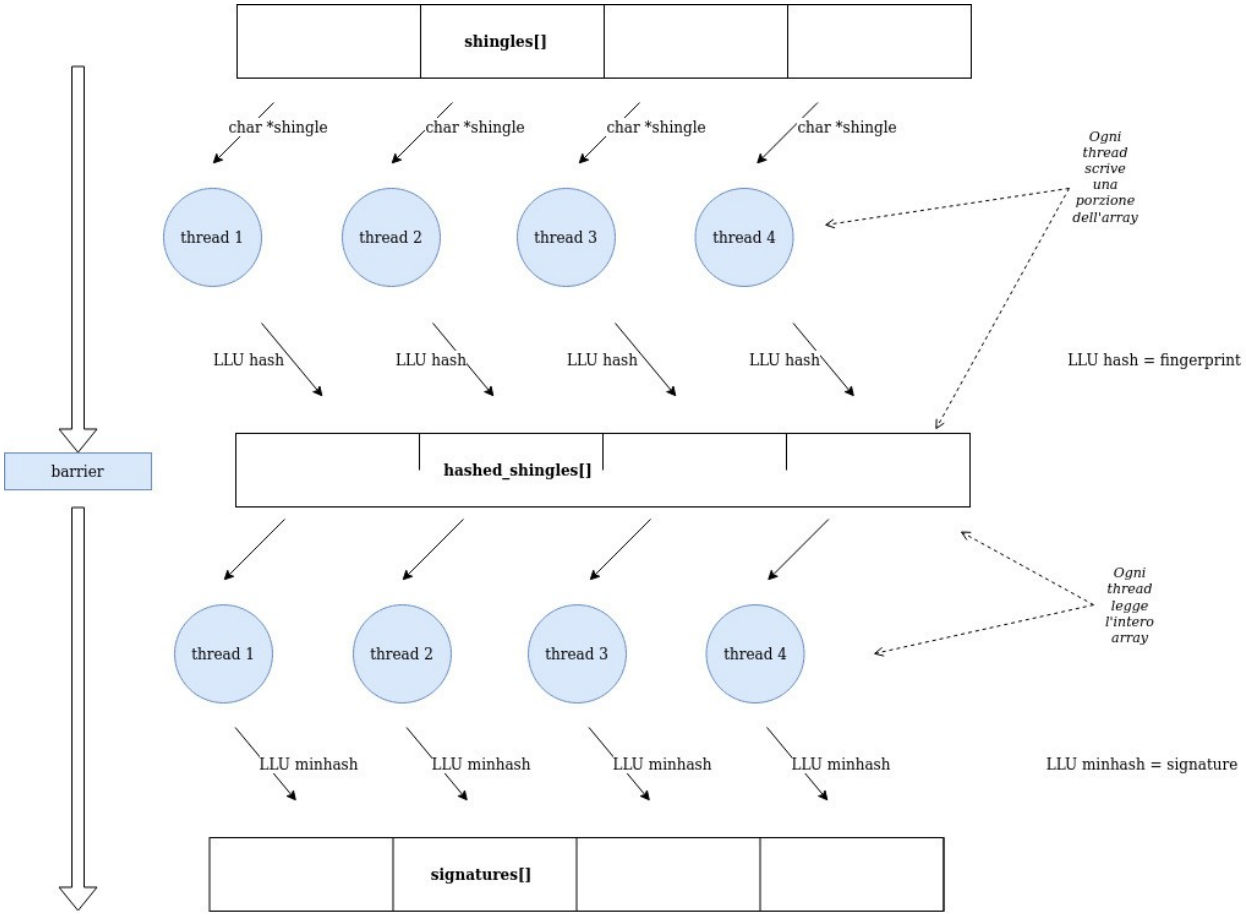
2.2 OpenMP

In OpenMP si è applicata una parallelizzazione dell’algoritmo seriale basata sui dati: si è cercato di sfruttare la propensione della libreria alla parallelizzazione dell’algoritmo seriale già studiato per essere diviso in piccoli task. Di questi abbiamo identificato quelli che potevano essere eseguiti in parallelo, sfruttando le strutture dati preesistenti come array e matrici. La comunicazione tra task avviene principalmente tramite condivisione di strutture dati. Questo riduce i problemi di dipendenze tra thread che lavorano suddividendosi equamente array e matrici. I problemi di agglomeration rimanenti sono stati risolti tramite le istruzioni critical, atomic, reduction, barrier.



1: Schema ad alto livello della parallelizzazione in OpenMP dell'algoritmo. Si può osservare come sono gestiti i thread annidati. La versione con Pthread è analoga.

Quasi nessuna funzione ha problemi di load-balancing perché i thread lavorano su array di dati equamente divisi, anche se nel main a ogni thread vengono assegnati dei files di diversa grandezza e questo può portare a una distribuzione non equa del carico dei dati. Inoltre in create_triplets ci possono essere problemi di load-balancing legati a un while che dipende da condizioni verificate a runtime.



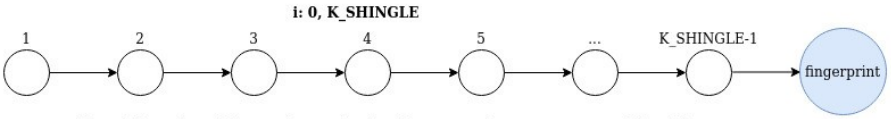
2: Schema della parallelizzazione sui dati della funzione get_signatures().

2.3 Pthreads

La parallelizzazione con Pthreads segue lo stesso approccio basato sui dati seguito con OpenMP. Con questa libreria per le sue peculiarità è stata definita un utilizzo più puntuale controllo dei fork dei sotto-thread e una più puntuale suddivisione delle sottoparti delle matrici su cui questi vanno a lavorare. Si è resa necessaria la creazione di strutture dati ad hoc per il passaggio degli argomenti per ogni thread.

hash_FNV_1a

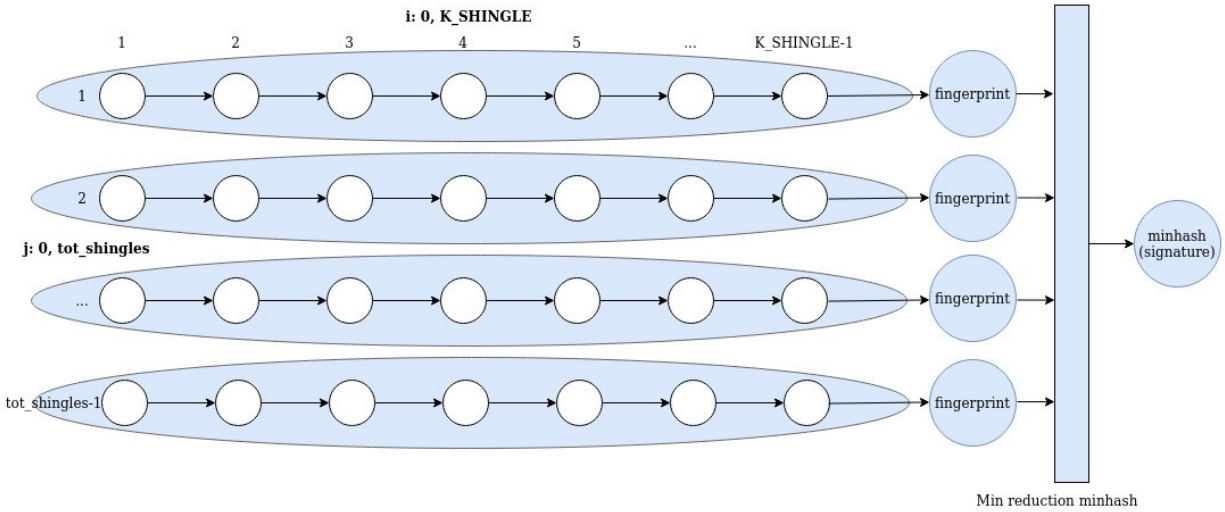
```
hash = FNV_offset_basis;

  for(int i=0; i<K_SHINGLE; i++){
     (*hash) *= prime;
     (*hash) ^= shingle[i];
  }
```



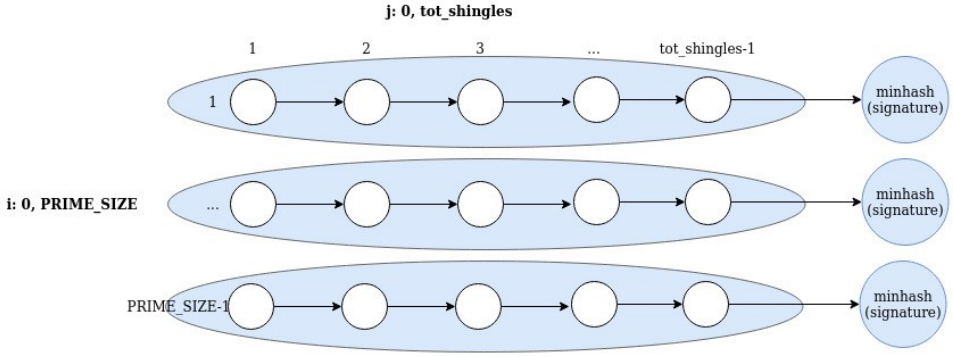
ogni dato è dipendente dal precedente, e l'ordine deve essere rispettato: non parallelizzabile

get_signatures

```
for(long j=0; j < tot_shingles; j++){
   hash_FNV_1a(shingles[j], &hash);
   hashed_shingles[j] = hash;
   if(hash < private_minhash)
      private_minhash = hash;
}
/*critical section*/
if (global_minhash > private_minhash)
   global_minhash = private_minhash;
```



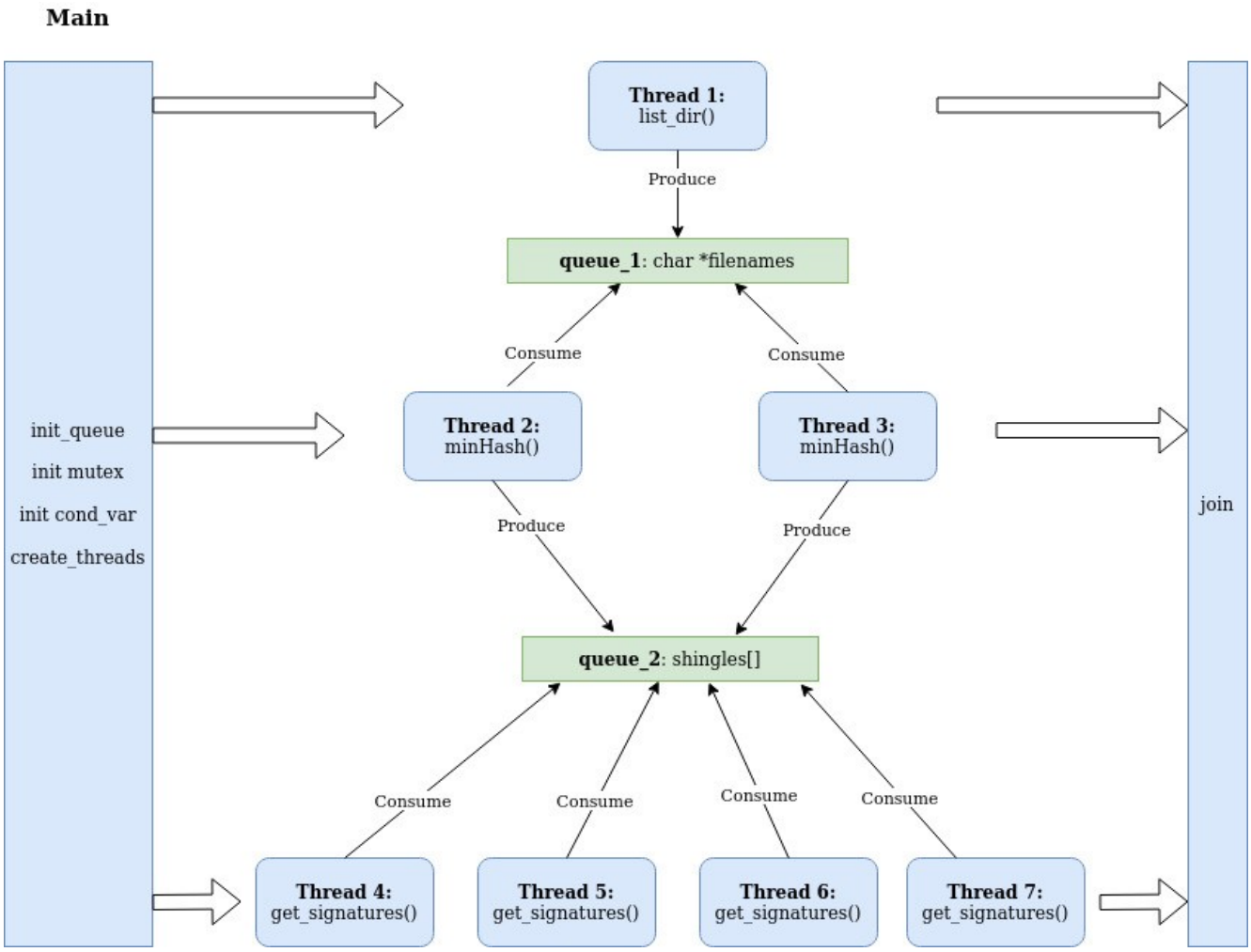

```
for(int i=0; i<PRIMES_SIZE; i++){
   minhash = MAX_LONG_LONG_U;
   for(int j=0; j<tot_shingles; j++){
      hash = hashed_shingles[j] ^ rands[i];
      if(hash < minhash)
         minhash = hash;
   }
   *(signatures+i+1)=minhash;
}
```



2.4 Produttore-Consumatore

Questa è una versione sperimentale che utilizza una combinazione delle librerie di Pthread e OpenMP per eseguire il programma. Pthread viene utilizzato per eseguire una parallelizzazione sui task, mentre OpenMP una sui dati, in particolare nel ricavare le signature.
La parallelizzazione sui task utilizza tre task per compiere la prima parte dell'algoritmo: il primo task si occupa di estrarre i nomi dei file con la funzione list_dir, il secondo task si occupa della lettura file e della creazione degli shingle e il terzo crea le signatures. La comunicazione tra questi tre task avviene per mezzo di due code. Il primo task è produttore nella prima coda, il secondo consuma dalla prima coda e produce nella seconda, il terzo consuma dalla seconda coda.
Per la sincronizzazione tra produttori e consumatori si usa una combinazione di condition variables e mutex. Ai task viene assegnato un numero variabile di thread proporzionale al costo computazionale e sempre inferiore al numero dei files.

Rispetto alle altre versioni sono stai integrati due moduli, prod_cons.c e queue.c dove sono definite nel primo le funzioni per gestire i thread consumatori e produttori e nel secondo la struttura dati della coda. Inoltre alcune funzioni del main sono state spostate nel file minhash.c.



Test

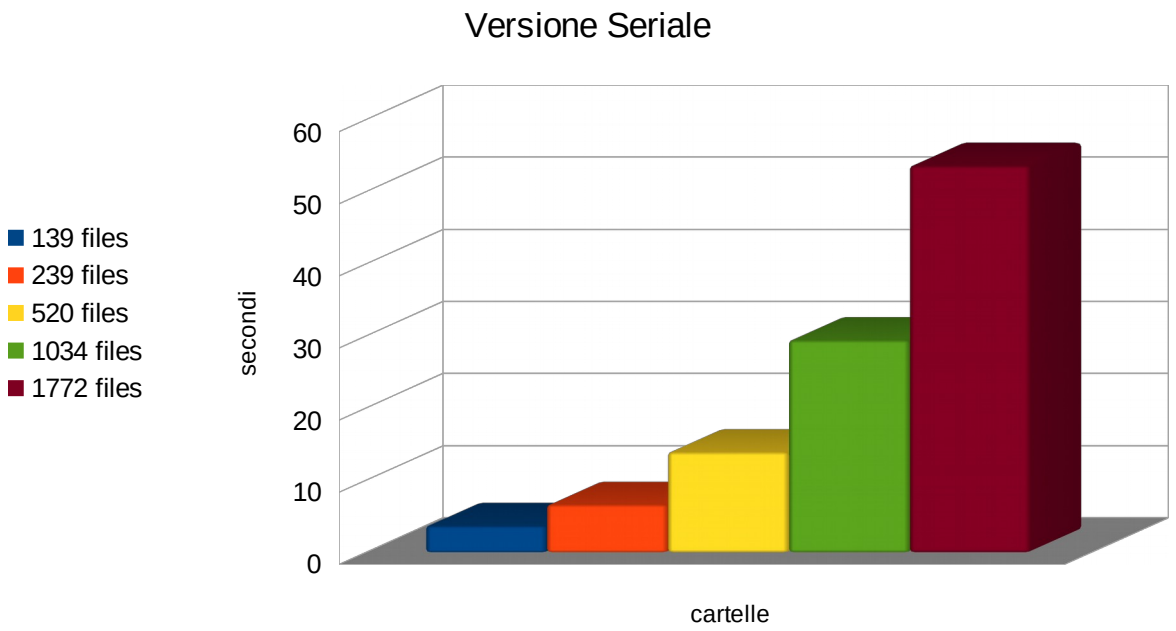
Per realizzare i test è stata creata una funzione che raccoglie i tempi di tutte le altre funzioni dell'algoritmo. Di ognuna di queste viene considerato il tempo peggiore, ovvero il thread che ha richiesto più tempo. A seconda della libreria usata si sono utilizzate funzioni specifiche per misurare il tempo, ma tutte equivalenti perchè basate sulla misurazione del wall time.

I valori dei tempi misurati vengono salvati in un file csv e in un file txt formattato per una rapida consultazione. Inoltre vengono salvati nel file similarity_results.txt i risultati in termini di similarità dell'algoritmo, da confrontare tra le varie versioni per controllarne la correttezza.

I test sono stati ripetuti per ogni versione dell'algoritmo su 5 cartelle con un numero sempre crescente di files (docs_verysmall, docs_small, docs_medium, docs_large, docs_verylarge), con una dimensione complessiva che all'incirca raddoppia sempre. Essendo il minhash principalmente utilizzato per file di piccole dimensioni ci siamo limitati, per poter garantire un'omogeneità nei test, a documenti di dimensioni tra i 2KB e i 38KB. Oltre i 1772 files il programma richiede una quantità di memoria superiore a quella disponibile sulla macchina utilizzata.

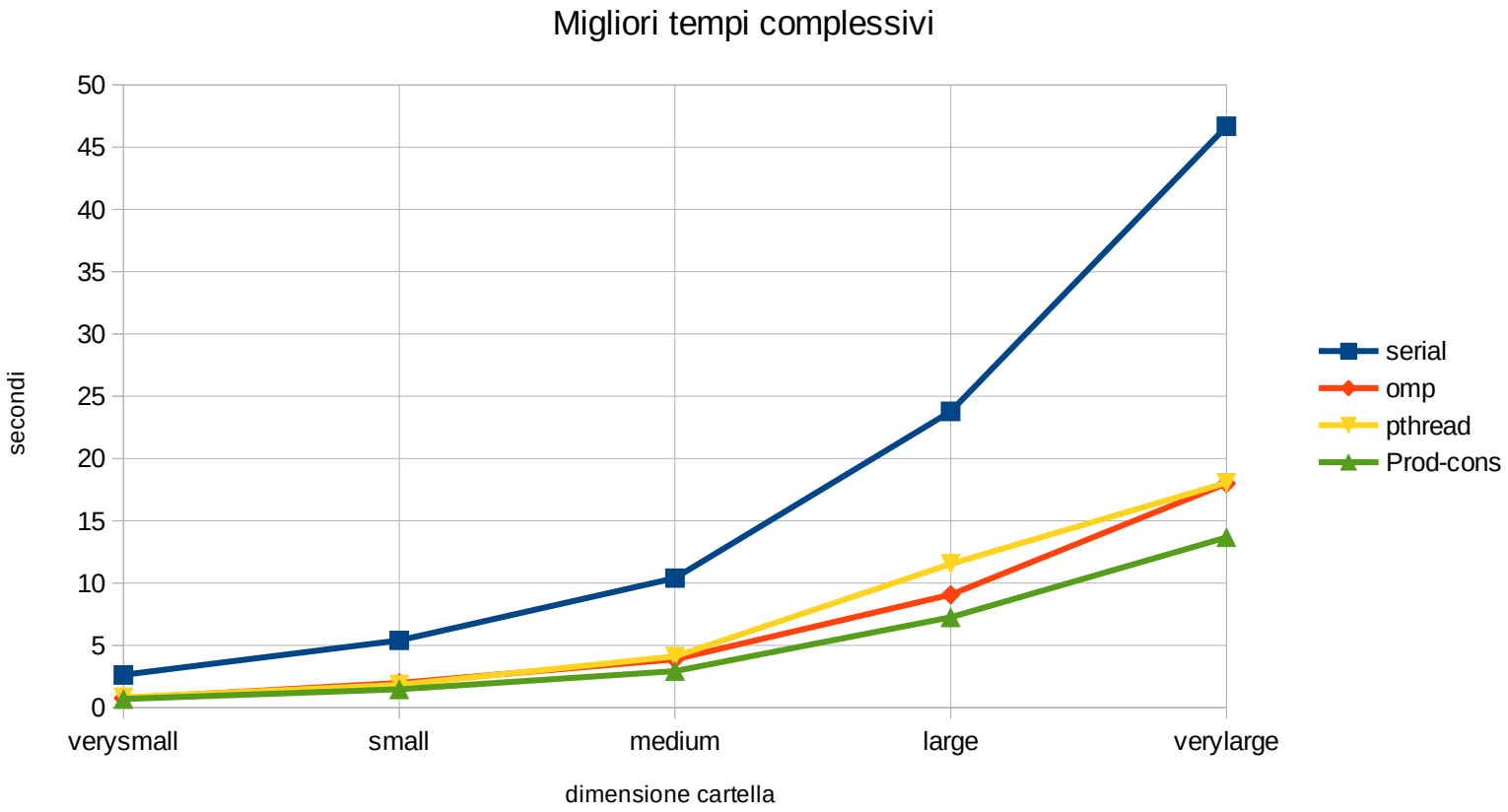
I test sono stati anche ripetuti con un numero variabile di threads, da 1 a 64, su una macchina dotata di 8 core. Vengono lanciati dallo script minhash.sh e il numero di thread viene passato da questo come parametro ai file eseguibili. La versione parallelizzata con OpenMP inoltre è stata testata modificando indipendentemente il numero dei thread delle funzioni principali e quello delle funzioni interne.

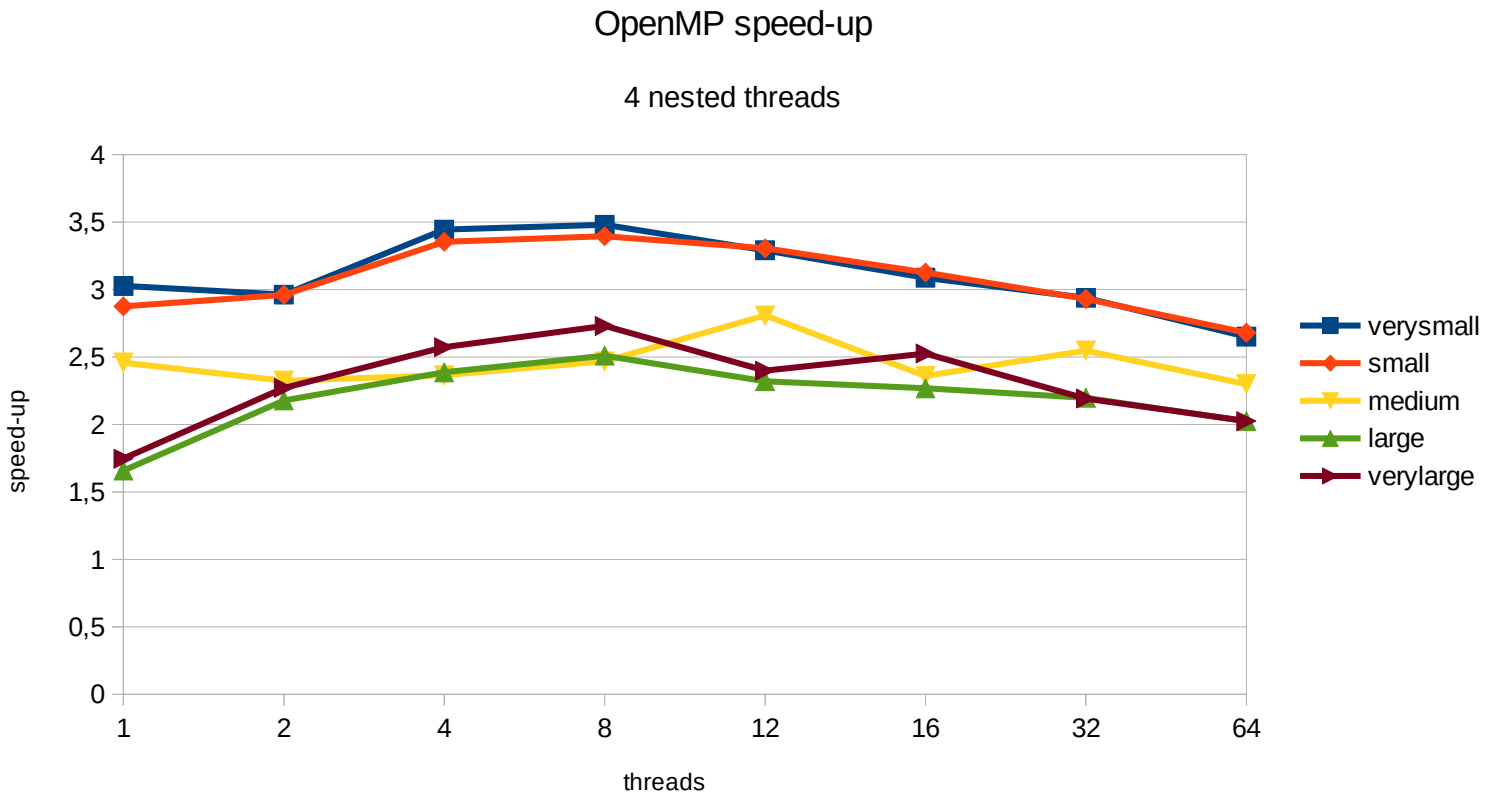
Anche la versione Produttore-Consumatore è stata testata con un numero variabile di thread, ripartiti tra produttori e consumatori a seconda del loro costo computazionale e tenendo conto degli 8 core a disposizione. Al primo produttore, che è molto più veloce rispetto agli altri task è sempre stato assegnato un solo thread. Al secondo e terzo task invece un numero crescente secondo questo schema: 1-1-2, 1-2-4, 1-3-6. Infine alle funzioni interne parallelizzate con OpenMP sono stati assegnati numeri fissi di thread.

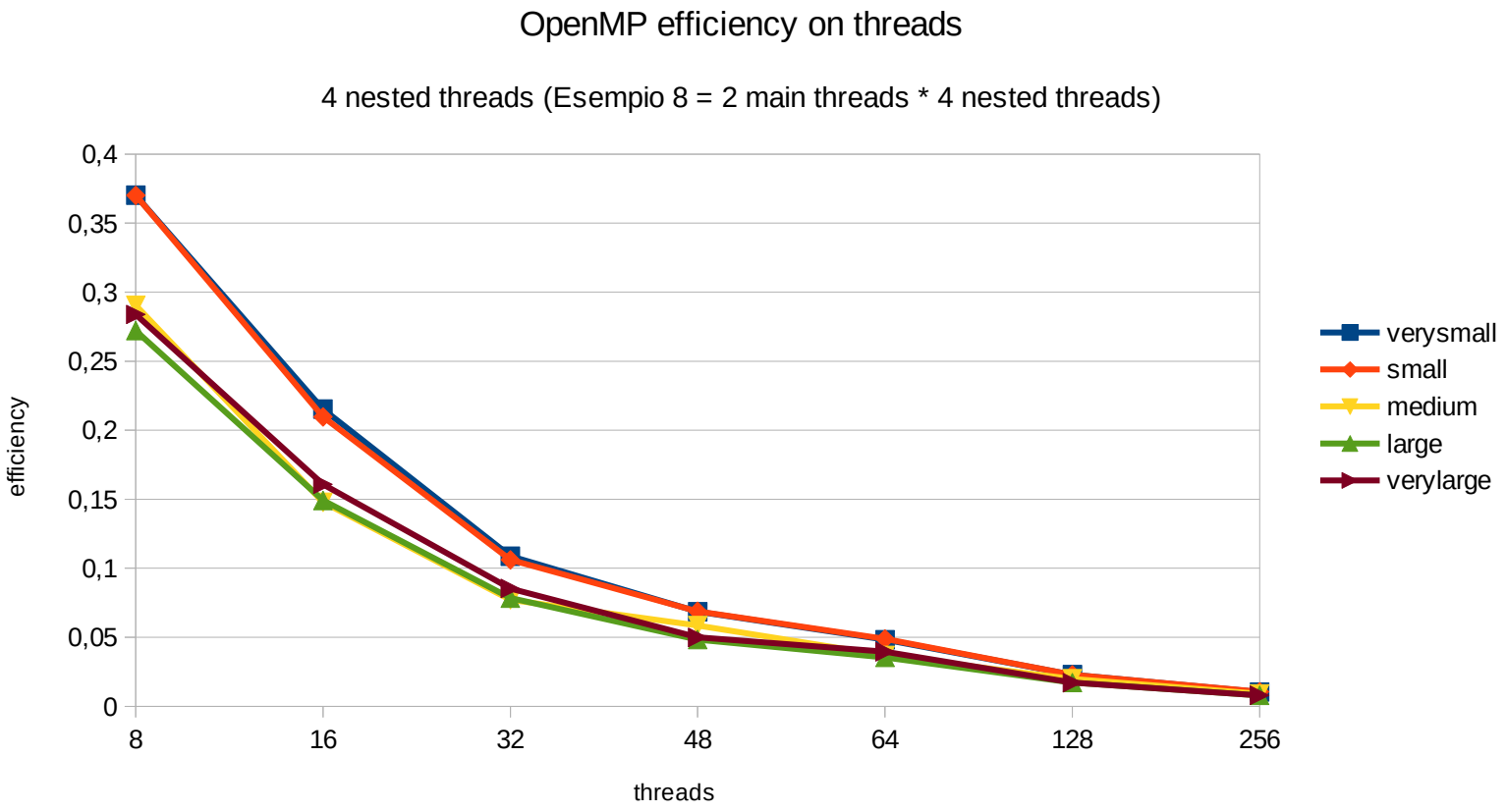


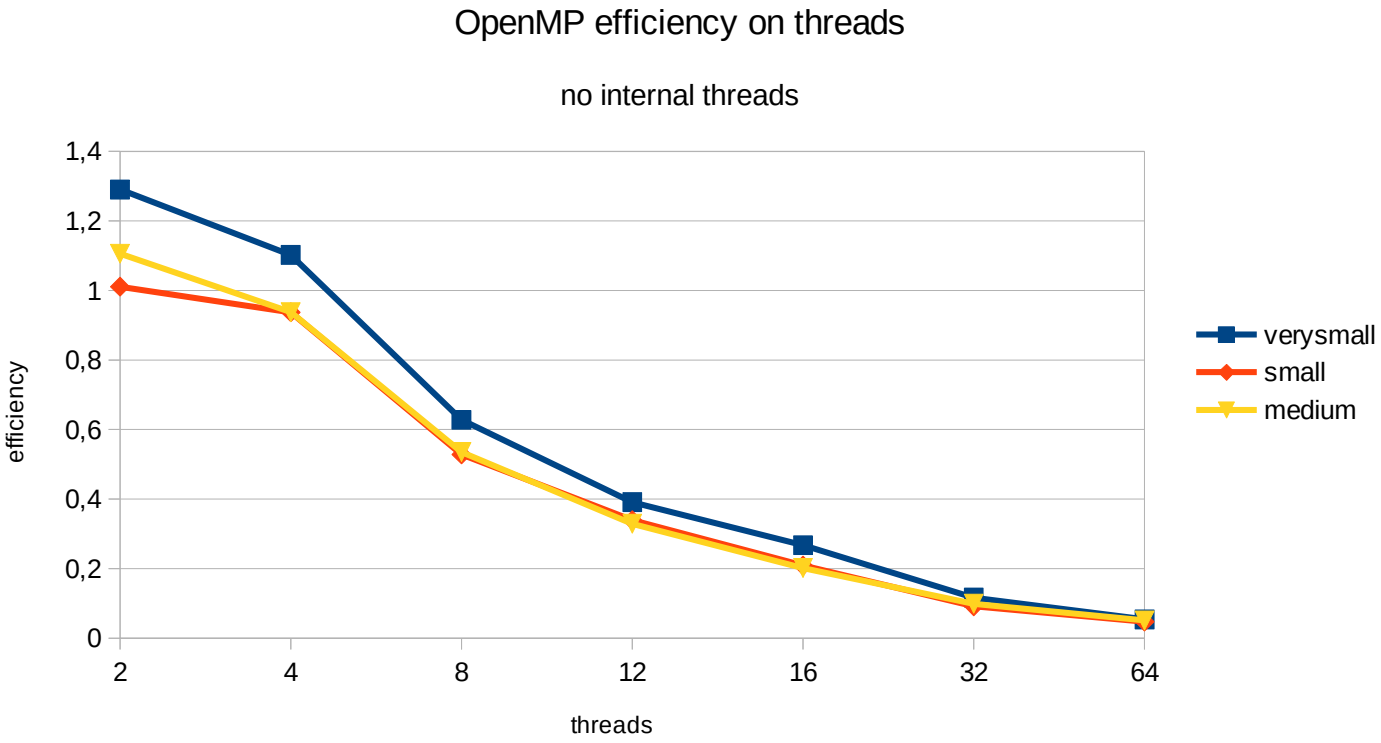
I tempi complessivi migliori per ogni versione espressi in secondi e relativo grafico.

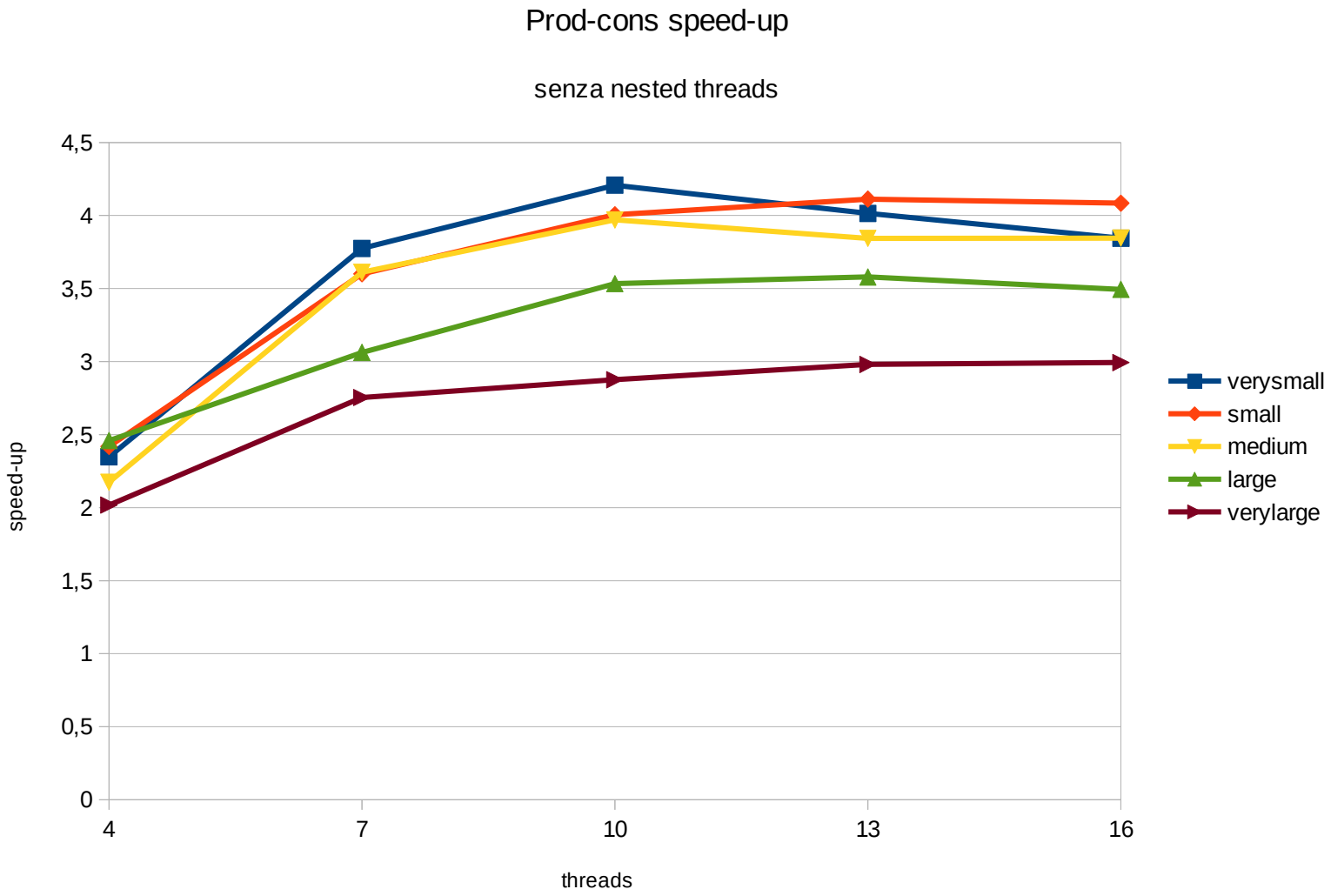
	verysmall	small	medium	large	verylarge
serial	2,6248	5,3966	10,3943	23,7746	46,6781
omp	0,7433	1,9806	3,8802	9,0643	18,0036
pthread	0,8207	1,8407	4,1068	11,5432	18,0566
Prod-cons	0,6823	1,4602	2,9347	7,2424	13,6676

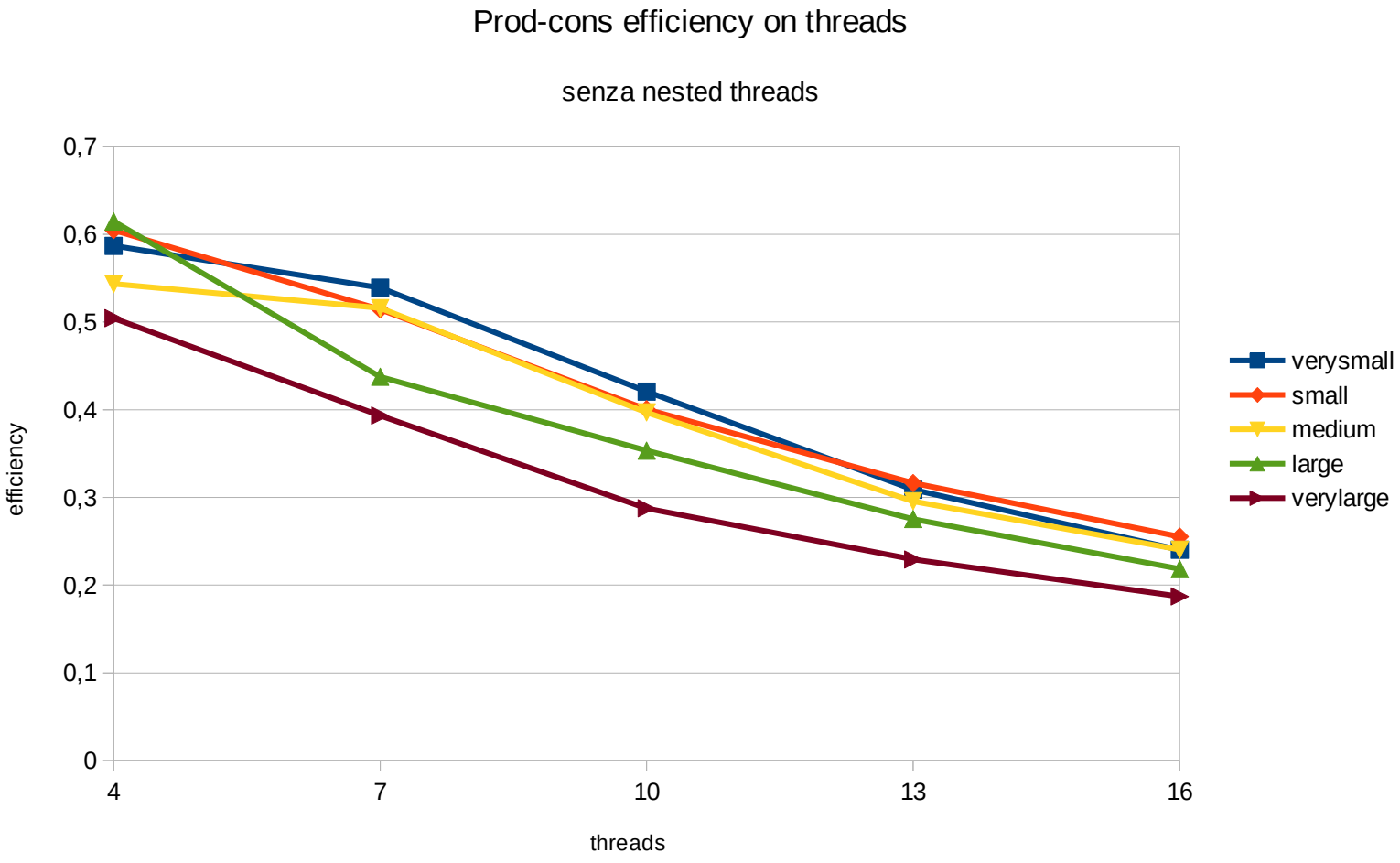


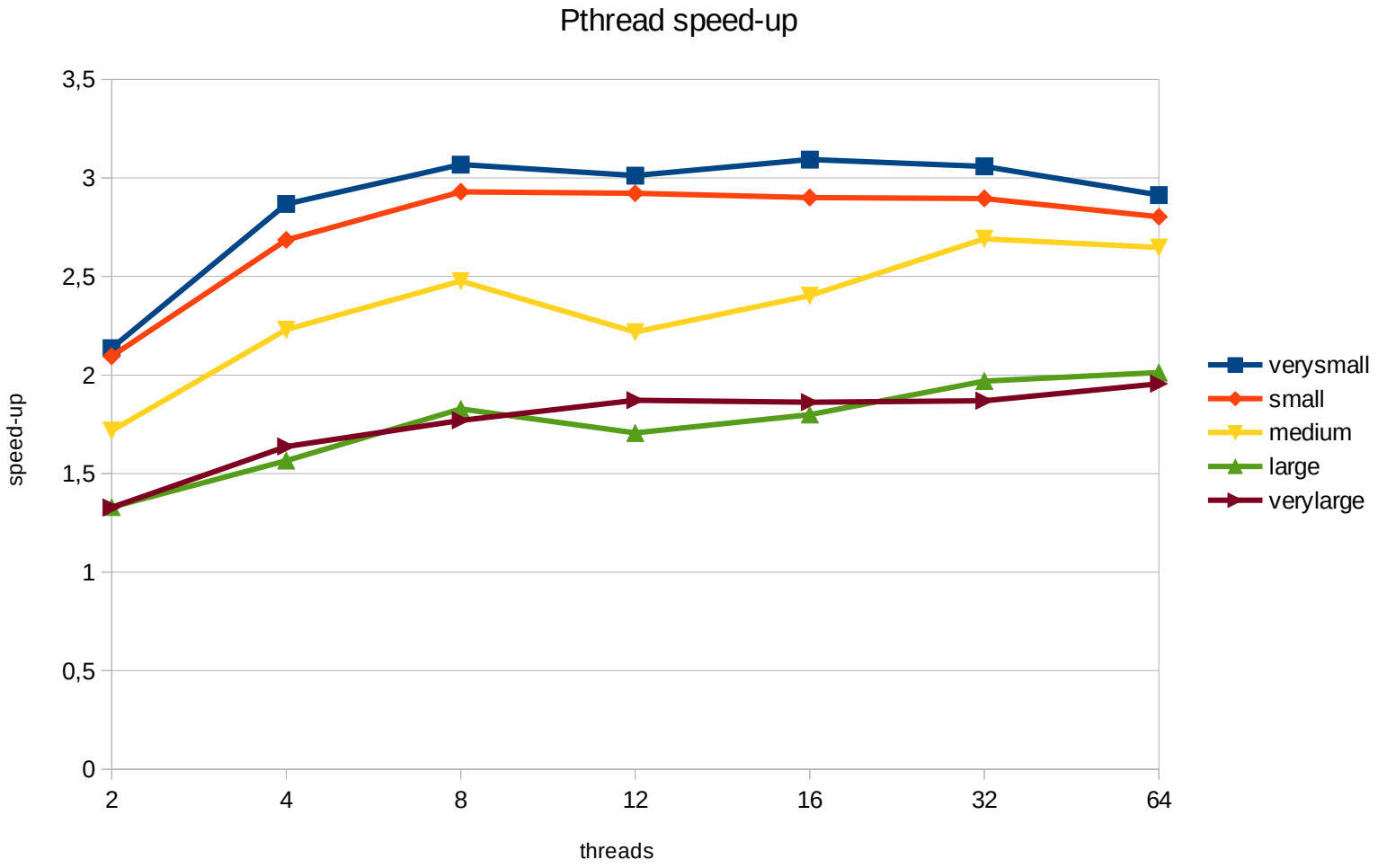

OpenMP speed-up
4 nested threads
speed-up
4
3,5
3
2,5
2
1,5
1
0,5
0
1
2
4
8
12
16
32
64
threads
verysmall
small
medium
large
verylarge

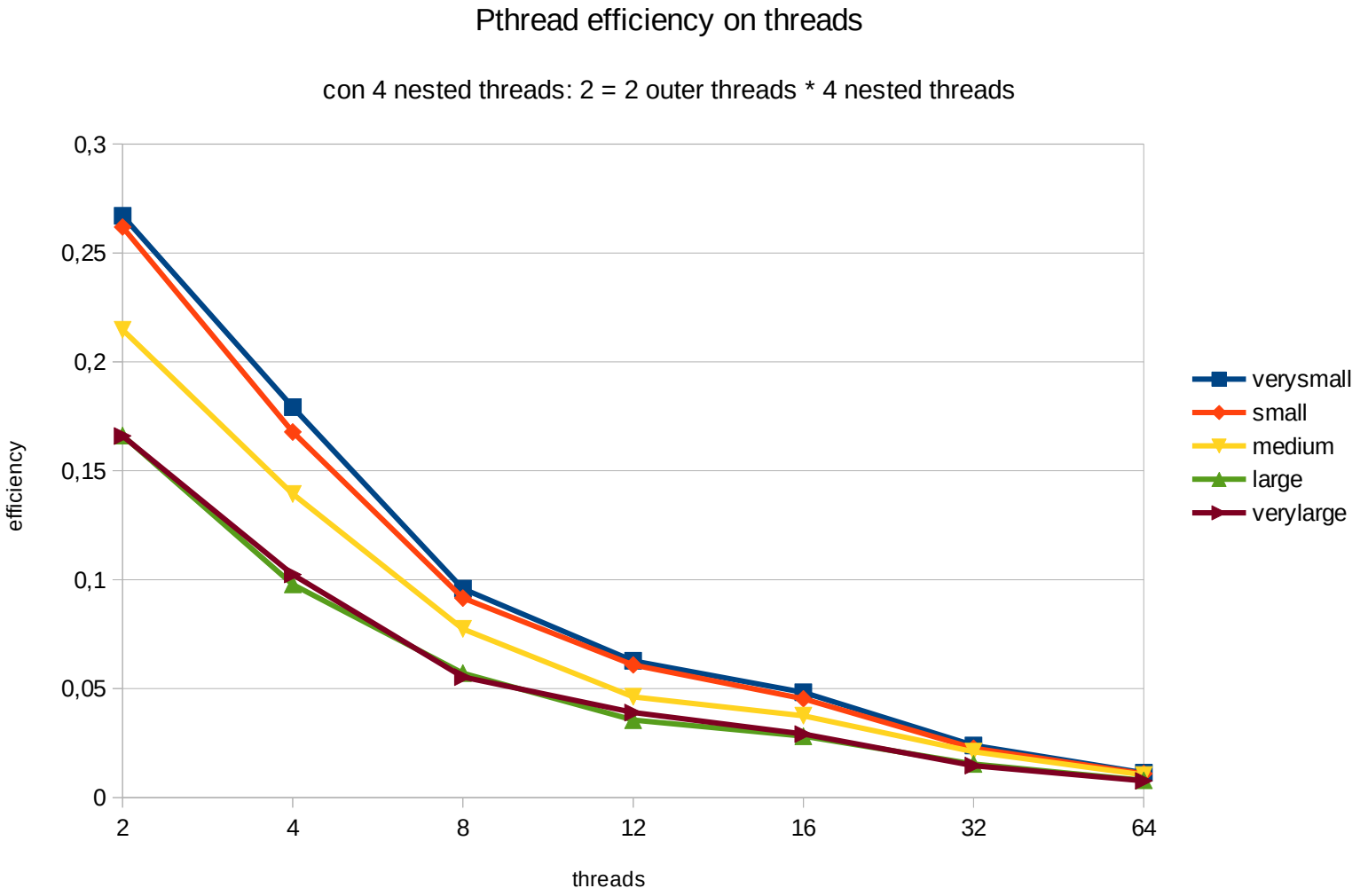

OpenMP efficiency on threads
4 nested threads (Esempio 8 = 2 main threads * 4 nested threads)
efficiency
0,4
0,35
0,3
0,25
0,2
0,15
0,1
0,05
0
8
16
32
48
64
128
256
threads
verysmall
small
medium
large
verylarge


OpenMP efficiency on threads
no internal threads
efficiency
1,4
1,2
1
0,8
0,6
0,4
0,2
0
2
4
8
12
16
32
64
threads
verysmall
small
medium


Prod-cons speed-up
senza nested threads
4,5
4
3,5
3
2,5
2
1,5
1
0,5
0
speed-up
4
7
10
13
16
threads
verysmall
small
medium
large
verylarge


Prod-cons efficiency on threads
senza nested threads
0,7
0,6
0,5
0,4
0,3
0,2
0,1
0
efficiency
4
7
10
13
16
threads
verysmall
small
medium
large
verylarge


Pthread speed-up
3,5
3
2,5
2
1,5
1
0,5
0
speed-up
2
4
8
12
16
32
64
threads
verysmall
small
medium
large
verylarge


Pthread efficiency on threads
con 4 nested threads: 2 = 2 outer threads * 4 nested threads
0,3
0,25
0,2
0,15
0,1
0,05
0
efficiency
2
4
8
12
16
32
64
threads
verysmall
small
medium
large
verylarge

References:

[1] https://en.wikipedia.org/wiki/MinHash
[2] https://it.wikipedia.org/wiki/Indice_di_Jaccard
[3] Identifying and Filtering Near-Duplicate Documents, Andrei Z. Broder
[4] http://www.dmf.unisalento.it/~scienze/Download/algebra_2new.pdf
[5] https://en.wikipedia.org/wiki/Fowler%E2%80%93Noll%E2%80%93Vo_hash_function