Valutazione della GBWT nel Mapping con *Giraffe*

e Analisi Comparativa con BWA-MEM2

UNIVERSITÀ DEGLI STUDI
DI SALERNO

di Kevin Paladino

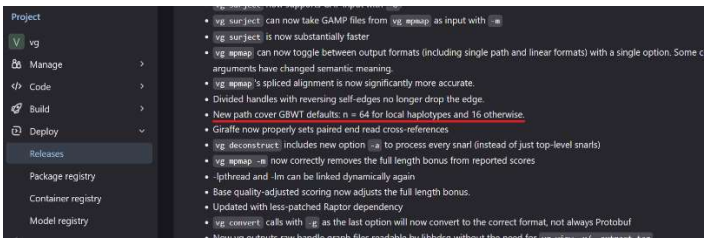
Sommario

Informazioni preliminari

L'obiettivo di questo lavoro è valutare in quali condizioni e in che misura l'indice **GBWT** (che rappresenta gli aplotipi nel Variation Graph) migliora l'allineamento delle letture di DNA; anche rispetto a un approccio lineare tradizionale.

Costruirò un Variation Graph utilizzando il cromosoma 6 del genoma umano GRCh38 e le varianti reali (phased) provenienti da dati VCF del progetto 1000 Genomi. Dopodiché eseguirò il mapping con **Giraffe** con più letture (vedi sotto) su una regione specifica del genoma umano (vedi ancora più sotto). Infine confronterò le prestazioni dell'allineamento in tre configurazioni della GBWT:

Configurazione standard: cioè con
1. **path-cover: n=64** (default value)



Configurazione a diversa densità aplotipica: versioni del grafo costruite con
2. **path-cover: n=16**
3. **path-cover: n=128**

ⓘ path-cover: è un insieme di percorsi che copre tutti i nodi del variation graph. Avere più percorsi nel path-cover dovrebbe ampliare il ventaglio di alternative disponibili per l'allineamento.

Prima di procedere, effettuerò un **Sanity Check**, utilizzando letture simulate direttamente dal riferimento (e solo sulla configurazione standard), per verificare il corretto funzionamento della pipeline.

Successivamente, la pipeline completa (tutte e tre le configurazioni) di *Giraffe* verrà eseguita due volte con letture differenti:
1. **Mapping sintetico**: allineamento di letture simulate "**haplotype-aware**", cioè basate su sequenze che riflettono la variabilità reale.
2. **Mapping reale**: allineamento di letture reali provenienti dal campione umano **HG002**, per la valutazione conclusiva.

Nota sulla scelta del campione HG002:
L'ho scelto perché voglio un uso realistico del pangenoma, ossia allineare **un nuovo individuo** che non fa parte della coorte scelta, infatti HG002 non fa parte del Progetto 1000 Genomi.

Infine, eseguirò un **allineamento lineare** (con le stesse letture reali) sul genoma di riferimento utilizzando **BWA-MEM**, in modo da ottenere un termine di confronto con l'approccio tradizionale.

La regione genetica analizzata è il Complesso Maggiore di Istocompatibilità (MHC/HLA), situato in chr6:28,477,797–33,448,354 (~4.97 Mb). Questa area contiene geni fondamentali per la risposta immunitaria ed è una delle più variabili dell'intero genoma umano, motivo per cui rappresenta un ottimo banco di prova per la valutazione di metodi di allineamento in regioni altamente polimorfiche.

Problema 1: Scelta della regione

Problema 1.1: Scarsa variabilità

Il metodo utilizzato da *vg* per costruire la GBWT e per popolare il grafo dipende fortemente dalla densità di varianti: in regioni molto uniformi, o con varianti troppo distanti tra loro, il grafo risultante è quasi lineare.

Problema 1.2: Limiti computazionali e Gestione della mole di dati

Prima dell'esecuzione finale documentata, ho effettuato numerosi test preliminari su regioni genomiche più ampie o su subset troppo grandi del genoma, con il risultato di saturare rapidamente la memoria disponibile o di produrre esecuzioni estremamente lente.

Installazione del *vg toolkit*

I seguenti comandi:

- installano *Miniconda* (una distribuzione leggera di *Anaconda*), che semplifica radicalmente l'installazione e la gestione di pacchetti software e degli ambienti in cui questi vengono eseguiti, risolvendo il problema delle compatibilità tra versioni e dipendenze.
- configurano i canali per pacchetti bioinformatici
- e creano un ambiente isolato con alcuni strumenti necessari all'analisi di pangenomi, in particolare il *vg toolkit*.

```
wget https://repo.anaconda.com/miniconda/Miniconda3-latest-Linux-x86_64.sh
```

```
bash Miniconda3-latest-Linux-x86_64.sh
> ENTER
> yes
> ENTER
```

Verificare la cartella esista:

```
ls ~/miniconda3/
```

```
nano ~/.bashrc
```

in fondo al file aggiungere:

```
export PATH="$HOME/miniconda3/bin:$PATH"
> CTRL + O
> ENTER
> CTRL + X
```

```
source ~/.bashrc
```

```
conda --version
```

```
conda tos accept --override-channels --channel https://repo.anaconda.com/pkgs/main
conda tos accept --override-channels --channel https://repo.anaconda.com/pkgs/r
```

```
conda activate
```

```
conda update -y conda
```

```
conda config --add channels conda-forge
conda config --add channels bioconda
conda config --set channel_priority strict
```

```
conda install -y mamba -n base -c conda-forge
```

```
mamba --version
```

```
mamba create -n pangenome -c conda-forge -c bioconda \
  vg samtools bcftools tabix htslib pbwt python=3.10 vcftools mummer -y
```

Lista di alcuni comandi utili

Attivare conda:
`conda activate`

Se conda activate non funziona:
`source ~/miniconda3/bin/activate`

Lista degli ambienti disponibili:
`conda env list`

Verificare informazioni sugli ambienti:
`conda info --envs`

Lista pacchetti installati:
`conda list`

Aggiornare tutti i pacchetti nell'ambiente:
`mamba update -all` oppure `conda update --all`

Per aggiornare solo il vg toolkit (se installato in un environment specifico, attivare prima l'environment)
`mamba update vg` oppure `conda update vg`

Cercare pacchetti disponibili:
```
conda search nome_pacchetto
mamba search nome_pacchetto
```

Attivare un ambiente specifico:
`conda activate pangenome`

Disattivare ambiente/conda
`conda deactivate`
se si è nell'ambiente *base*, allora il comando disattiva interamente conda, altrimenti si esce dall'ambiente attivato in precedenza

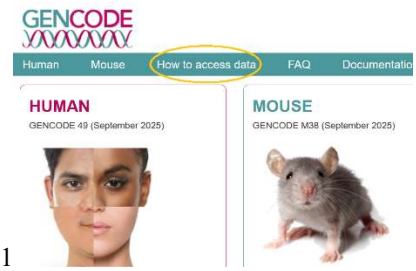
Fase 1: “Raccolta dati”

Creazione dell’ambiente di lavoro

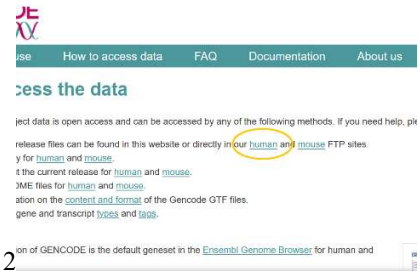
```
mkdir -p ~/tirocinio/data && cd ~/tirocinio/data
```

Download del genoma di riferimento umano: ossia delle sequenze nucleotidiche (cioè le successioni ordinate di: A, T, C, G) di tutti i cromosomi
Fonte: *The GENCODE Project: Encyclopædia of genes and gene variants*

1

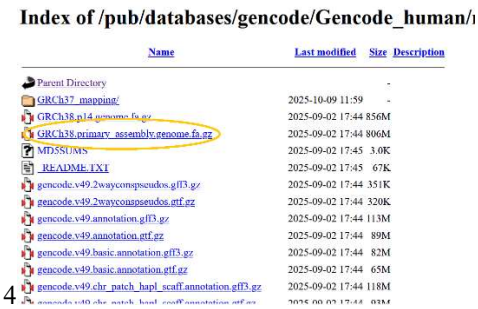


2



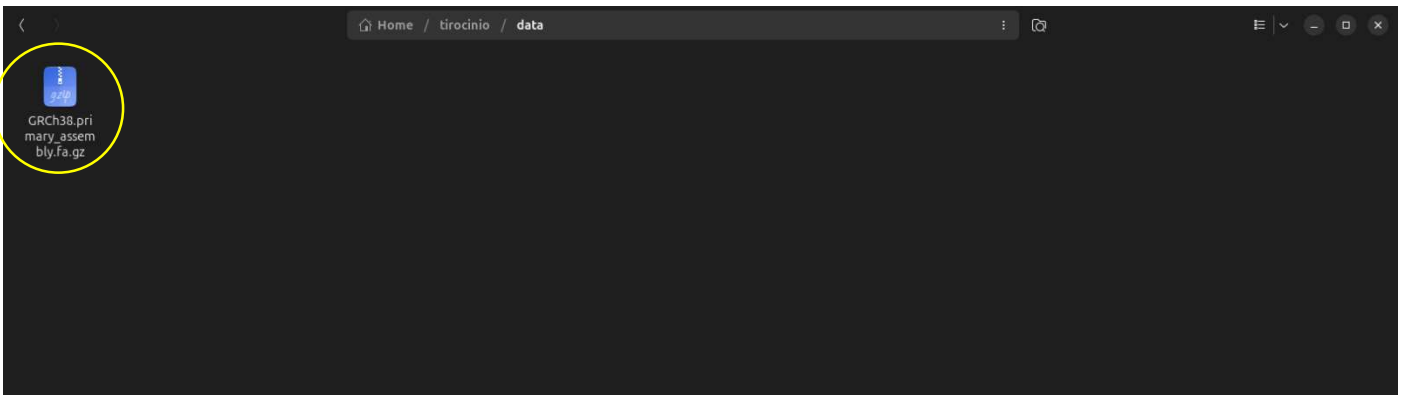
3

4




```
wget -O GRCh38.primary_assembly.fa.gz
ftp://ftp.ebi.ac.uk/pub/databases/gencode/Gencode_human/release_49/GRCh38.primary_assembly.genome.fa.gz
```

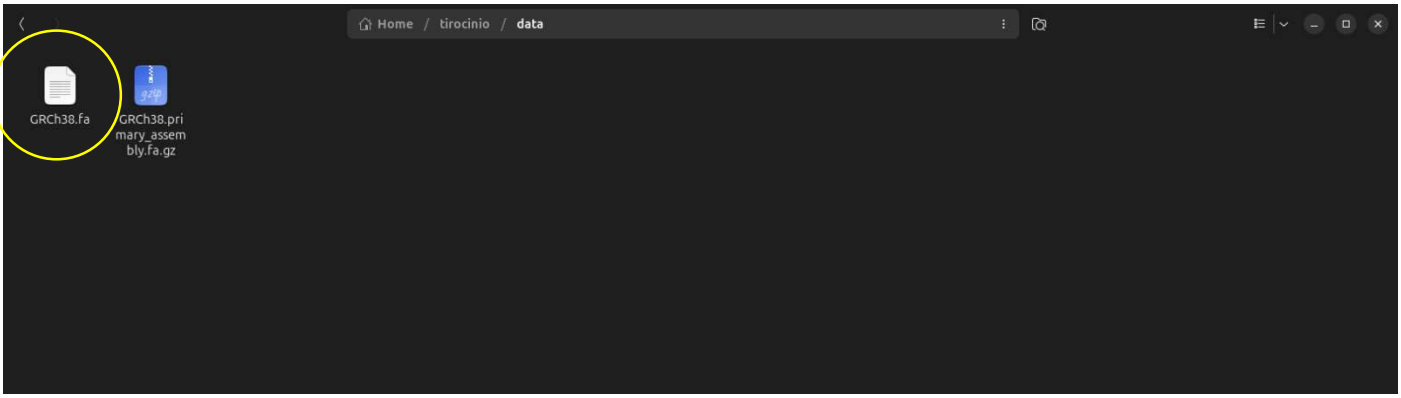
`-O`: nome del file di output “…”
~ 845.6 MB



Estrazione del file FASTA

```
gunzip -c GRCh38.primary_assembly.fa.gz > GRCh38.fa
```

`-c`: mantieni l’originale
~ 3.2 GB



Struttura di un file *.fa*

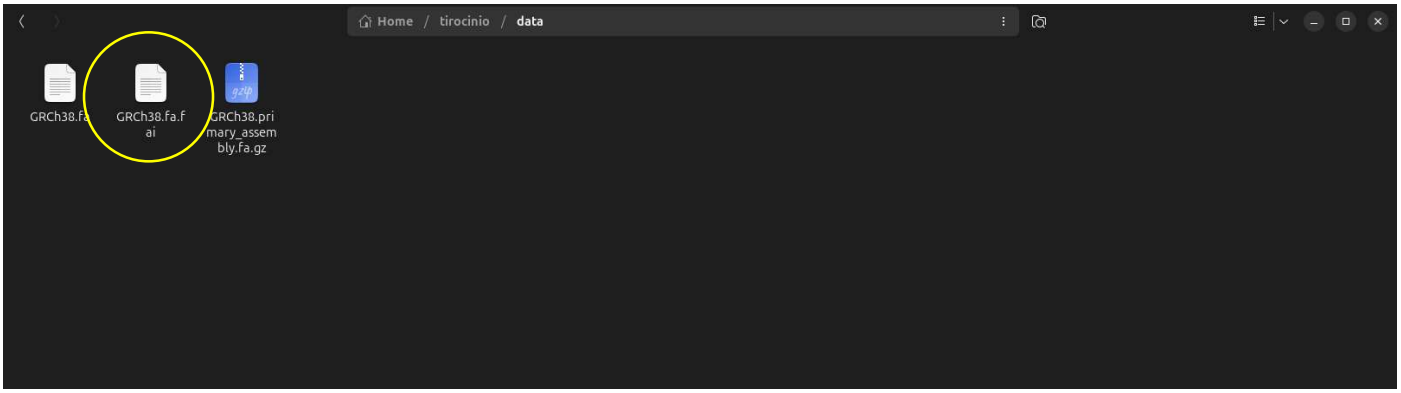
>header: inizia con il simbolo > e contiene metadati identificativi
sequenza: contiene caratteri che rappresentano nucleotidi o amminoacidi

`head -n file.fa`	Mostra le prime *n* righe del file .fa/.fai
`grep ">" file.fa`	Mostra tutti i cromosomi/contig presenti

Indicizzazione del file FASTA: genera un indice che permette di estrarre rapidamente sequenze da cromosomi o regioni specifiche senza leggere l'intero file

```
samtools faidx GRCh38.fa
```

`samtools`: suite di strumenti per manipolare dati genomici
`faidx`: funzione di samtools specifica per lavorare su file FASTA (*.fa*), in questo caso crea il suddetto file di indice (*.fai*)
~ 6.5 kB



Struttura di un file *.fai*

Nome sequenza	*Lunghezza*	*Offset*	*Basi per linea*	*Byte per linea*
chr1	248956422	8	60	61
chr2	242193529	253105712	60	61
…	…	…	…	…

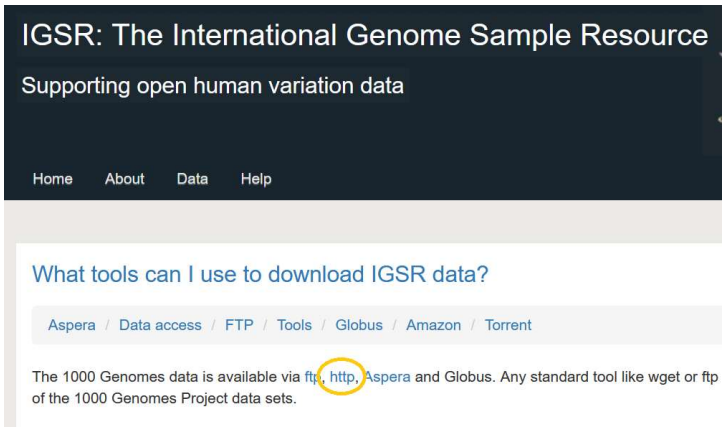
Estrazione di una regione genomica specifica dal riferimento

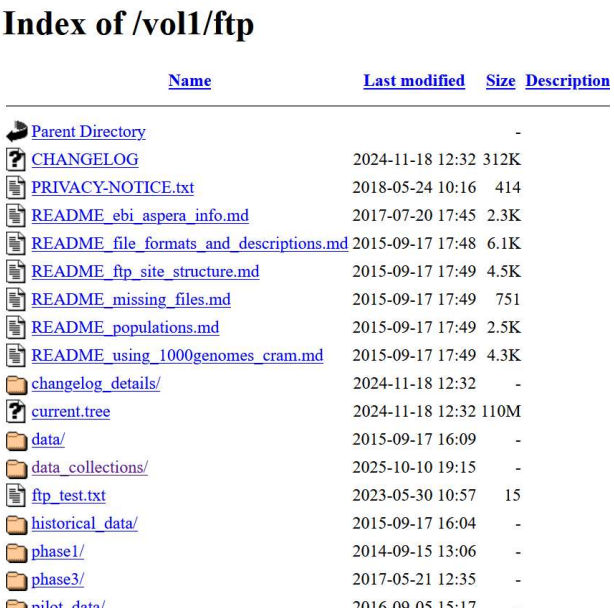
```
samtools faidx GRCh38.fa chr6:28477797-33448354 > MHC.ref.fa
```

Anche se la funzione è la stessa del comando precedente, poiché l'indice esiste già, il comando lo utilizza per un accesso rapido alla regione richiesta, ed estrae la sequenza in un nuovo file (quindi per uno scopo differente).
~ 5.1 MB

Il suddetto comando ci porterà ad affrontare il Problema 2.

Download dei dati VCF del cromosoma 6: tutte le varianti genetiche scoperte nel cromosoma 6
Fonte: *1000 Genomes (The International Genome Sample Resource)*
Via web accedo alla piattaforma *https://www.internationalgenome.org/faq/what-tools-can-i-use-to-download-igsr-data* e clicco su "http" (*https://ftp.1000genomes.ebi.ac.uk/vol1/ftp/*)



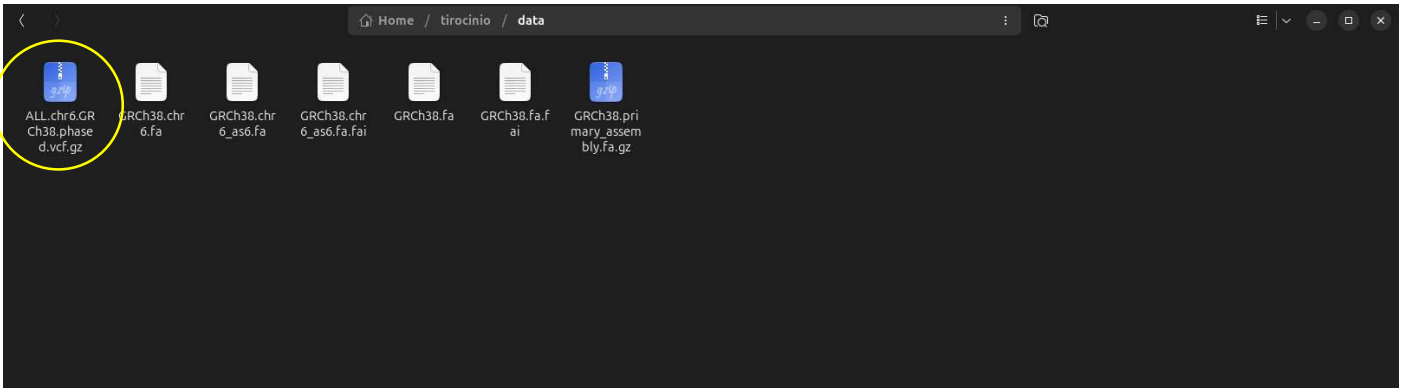


Seguo il percorso: */data_collections/1000_genomes_project/release/20190312_biallelic_SNV_and_INDEL/*

Link finale:
https://ftp.1000genomes.ebi.ac.uk/vol1/ftp/data_collections/1000_genomes_project/release/20190312_biallelic_SNV_and_INDEL/

```
wget -O ALL.chr6.GRCh38.phased.vcf.gz
http://ftp.1000genomes.ebi.ac.uk/vol1/ftp/data_collections/1000_genomes_project/release/20190312_biallelic_SNV_and_INDEL/ALL.chr6.shapeit2_integrated_snvindels_v2a_27022019.GRCh38.phased.vcf.gz
```

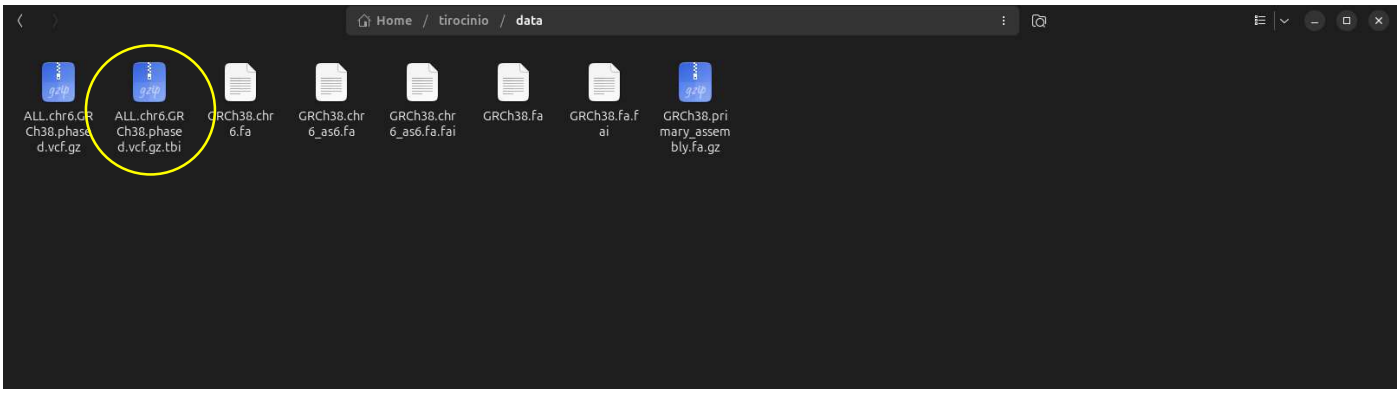
~ 863.7 MB



Download dell'indice del file VCF: necessario per accedere rapidamente a qualsiasi regione del cromosoma 6

```
wget -O ALL.chr6.GRCh38.phased.vcf.gz.tbi
http://ftp.1000genomes.ebi.ac.uk/vol1/ftp/data_collections/1000_genomes_project/release/20190312_biallelic_SNV_and_INDEL/ALL.chr6.shapeit2_integrated_snvindels_v2a_27022019.GRCh38.phased.vcf.gz.tbi
```

~ 161.5 kB



Ogni individuo possiede due copie di ciascun cromosoma autosomico: una ereditata dal padre e una dalla madre. Queste due copie non sono identiche: differiscono per molte varianti genetiche.
Quando analizziamo il genoma, queste varianti possono essere rappresentate in due modi

Genoma NON fasato (unphased)
Nel VCF non fasato conosciamo tutte le varianti presenti in una persona, ma non sappiamo su quale dei due cromosomi si trova ciascuna di esse. Di conseguenza:

- non possiamo ricostruire le due sequenze reali dei cromosomi,
- non conosciamo come le varianti si combinano realmente tra loro.

Genoma fasato (phased)
Nel VCF fasato le varianti sono assegnate ai due cromosomi separatamente.
Significa che sappiamo quali varianti appartengono all'aplotipo materno e quali a quello paterno.
Con il phasing possiamo quindi:

- ricostruire i due aplotipi,
- sapere quali varianti compaiono insieme sulla stessa copia del cromosoma,
- ottenere una rappresentazione molto più realistica della diversità genetica di un individuo.

Un mapper come Giraffe cerca di allineare ogni lettura seguendo un percorso plausibile nel grafo.

- Se gli aplotipi sono noti (phased), abbiamo percorsi reali, cioè sequenze che esistono davvero in qualche individuo.
- Se il genoma non è fasato, le varianti sono solo punti sparsi e non sappiamo come collegarle: è molto più difficile ricostruire un cammino biologicamente corretto.

Estrazione delle varianti scoperte nella regione MHC

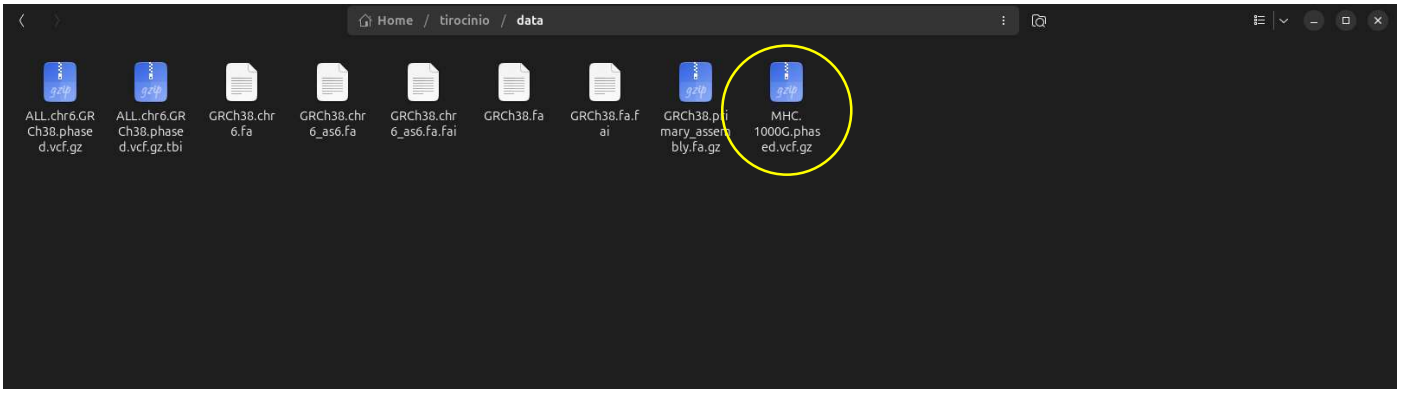
```
tabix -h ALL.chr6.GRCh38.phased.vcf.gz 6:28477797-33448354 | bgzip -c > MHC.1000G.phased.vcf.gz
```

`tabix`: strumento complementare a faidx di samtools, ma con questo si lavora con file in formato *.vcf*.
In questo caso, poiché l'indice già esiste (l'abbiamo scaricato apposta) e poiché ho specificato la regione, il comando non fa altro che estrarla.
`-h`: opzione che include anche le intestazioni del file VCF (cioè le righe che iniziano con ## e la riga #CHROM POS ID ...), altrimenti tabix estrarrebbe solo le varianti

`|`: prende l'output di tabix e lo passa a…

`bgzip`: che comprime l'output in formato BGZF (*.vcf.gz*), adatto all'indicizzazione
`-c`: scrivi l'output su stdout piuttosto che creare un file

Infine reindirizza l'output che arriva da bgzip in un nuovo file.
~ 48.6 MB



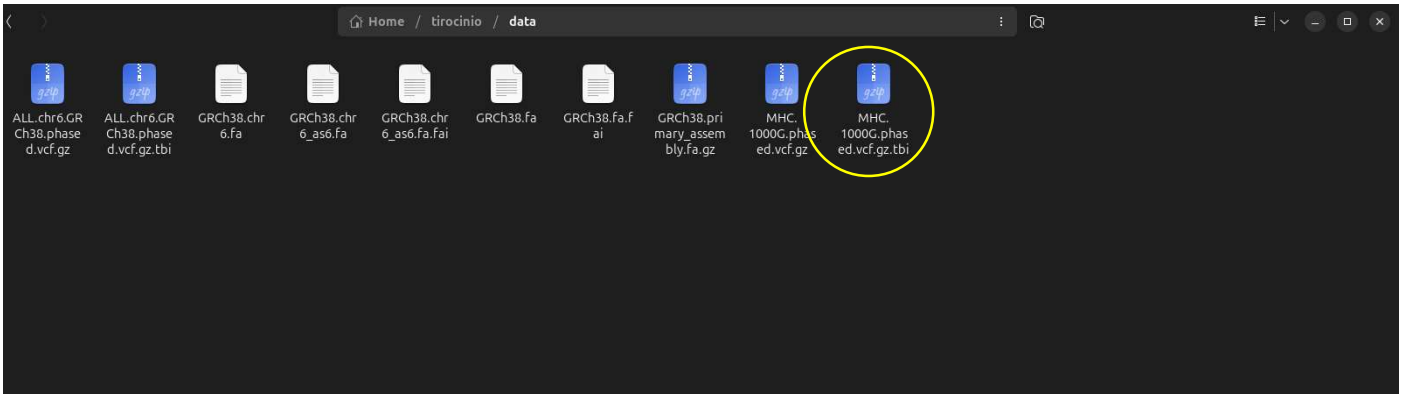
Indicizzazione del nuovo file VCF: crea un nuovo file di indice (*.tbi*) per il file VCF della regione MHC estratta

```
tabix -p vcf MHC.1000G.phased.vcf.gz
```

Analogamente a faidx, adesso creiamo l'indice della regione estratta, senza specificare null'altro.

`-p vcf`: Specifica che il file è in formato VCF

~ 3.6 kB



Fase 2.

Problema 2: mismatch dei nomi di contig tra FASTA e VCF

Poiché:

- Il nostro FASTA ha come header: `>chr6:28477797-33448354`
- Il VCF ha solamente: `6`
- non c'è modo per *vg* di capire che sono la stessa cosa, perché:
  - cambia il nome
  - cambia il sistema di coordinate: nell'ultimo FASTA, le posizioni vanno da 1 a ~5 Mb, nel VCF da ~28 Mb a ~33 Mb.

succede che il comando `vg autoindex` restituisca il seguente errore:

Due soluzioni alternative:

```
[vg::Constructor] Error: Reference contig "6" in VCF not found in FASTA.
```

Onde evitarlo ho trovato due soluzioni alternative:

- estrarre e lavorare sul cromosoma intero (ma sempre con focus sulla regione scelta), che tuttavia costa molta RAM e **pazienza** e per il quale si rischia il
  Problema 3: Saturazione memoria e Crash del Sistema Operativo
- rendere coerenti nomi e coordinate rinominando il vcf e riscrivendo le coordinate locali, usando altri strumenti, come *bcftools norm*, ecc. (soluzione per nulla banale da applicare, in quanto comporta tanti altri possibili errori da gestire, tra cui il
  Problema 4: Zero Varianti
  (`No paths other than selected reference(s) found in the graph, so no alt alleles can be generated`), per il quale non ho trovato soluzione.

Adoperiamo la prima soluzione:

invece del comando:

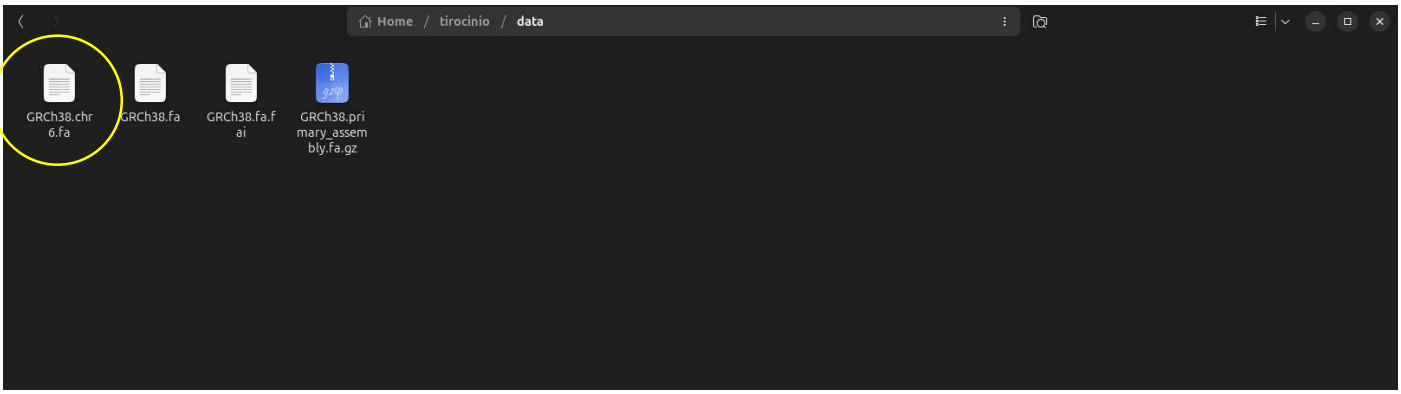
```
samtools faidx GRCh38.fa chr6:28477797-33448354 > MHC.ref.fa
```

Eseguiamo:

1)

```
samtools faidx GRCh38.fa chr6 > GRCh38.chr6.fa
```

~ 173.7 MB



2)

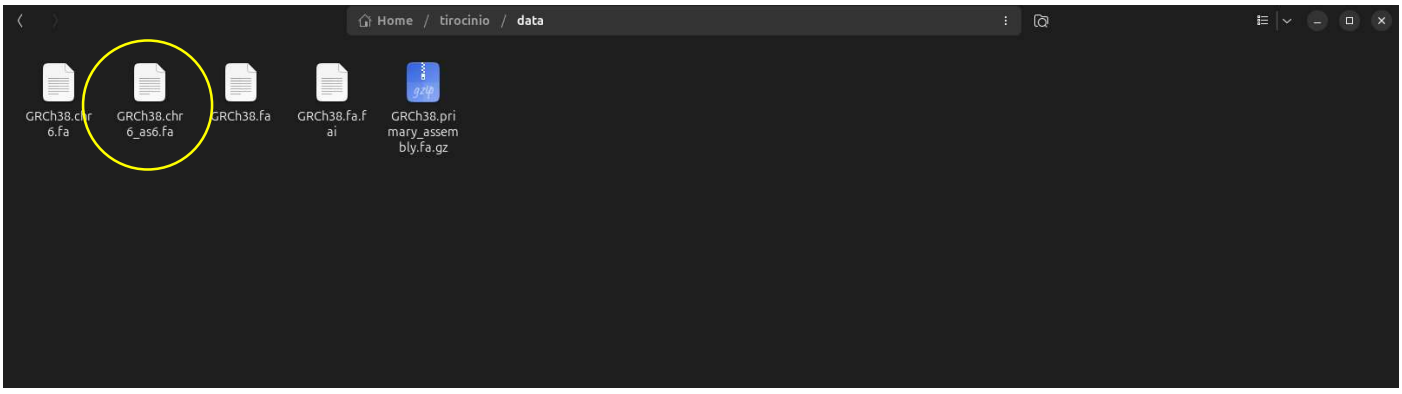
```
( echo ">6"; tail -n +2 GRCh38.chr6.fa ) > GRCh38.chr6_as6.fa
```

In un nuovo file FASTA inserisci:

— >6

— tutto il contenuto del FASTA originario, ma dalla riga 2 in poi

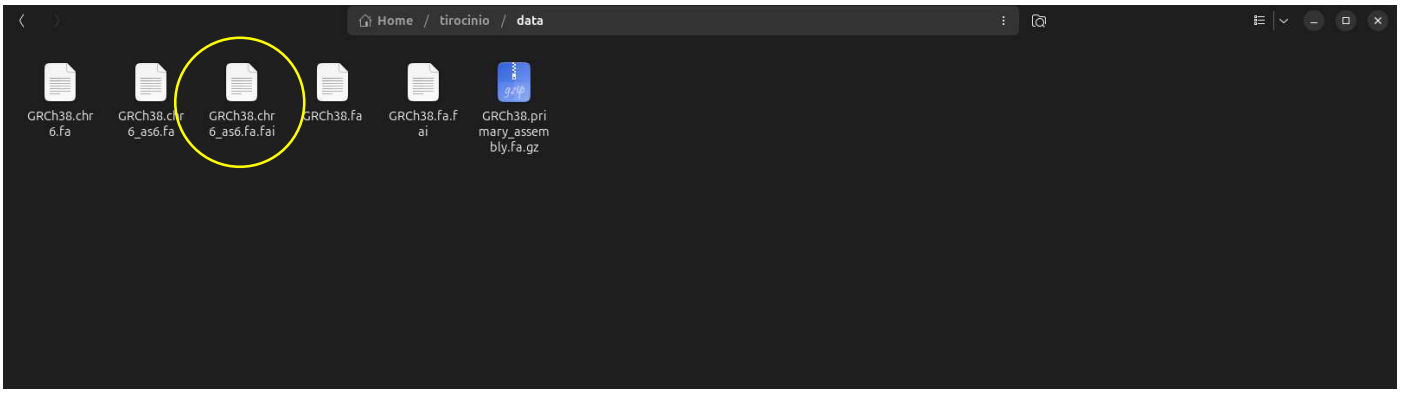
~ 173.7 MB



3)

```
samtools faidx GRCh38.chr6_as6.fa
```

~ 20 bytes



Verifica

```
head -n1 GRCh38.chr6_as6.fa
```

output: >6

```
tabix -l MHC.1000G.phased.vcf.gz | head
```

output: 6

OK.

Per riprendere la lettura, cliccare qui.

Fase 2: “Creazione del grafo pangenomico”

Preparazione dell’ambiente per l’indicizzazione

```
cd ..
mkdir -p work
```

Problema 5: pipeline manuale

All’inizio ho tentato di costruire il grafo seguendo una pipeline manuale basata su comandi “superati” come vg construct, vg index, vg gbwt, ecc.

Sebbene questa procedura sia ancora possibile, risulta oggi molto più complessa e incline a errori. La documentazione più recente di *vg* raccomanda invece l’uso del workflow integrato vg autoindex, che automatizza la costruzione del grafo, della GBWT e dell’indice Giraffe

Costruzione dell'indice per l'allineamento con vg giraffe

```
vg autoindex --workflow giraffe \
  -r data/GRCh38.chr6_as6.fa \
  -v data/MHC.1000G.phased.vcf.gz \
  -p work/MHC \
  --threads 16
```

`vg autoindex --workflow giraffe`: Lancia il workflow preconfezionato per costruire TUTTI gli indici necessari a *vg giraffe* (per short reads).

```
[vg autoindex] Executing command: vg autoindex --workflow giraffe -r data/GRCh38.chr6_as6.fa -v data/MHC.1000G.phased.vcf.gz -p work/MHC --threads 16
[IndexRegistry]: Checking for phasing in VCF(s).
[IndexRegistry]: Chunking inputs for parallelism.
[IndexRegistry]: Constructing VG graph from FASTA and VCF input.
[IndexRegistry]: Constructing GBWT from VG graph and phased VCF input.
[IndexRegistry]: Stripping allele paths from VG.
[IndexRegistry]: Constructing XG graph from VG graph.
[IndexRegistry]: Downsampling full GBWT.
[IndexRegistry]: Constructing GBZ.
[IndexRegistry]: Constructing distance index for Giraffe.
[IndexRegistry]: Constructing minimizer index and associated zipcodes.
        use parameters -k 29 -w 11
```

*MHC.giraffe**.gbz** (~ 156.7 MB)*: È il file principale: contiene il grafo pangenomico più le informazioni sui cammini (haplotipi o path-cover). Dentro ci sono:

- La sequenza di riferimento e le varianti (SNP, indel, ecc.)
- La GBWT: l’indice dei cammini (in questo caso, i 64 path-cover).
- La GBWTGraph: la parte “grafica” vera e propria (i nodi e le connessioni).

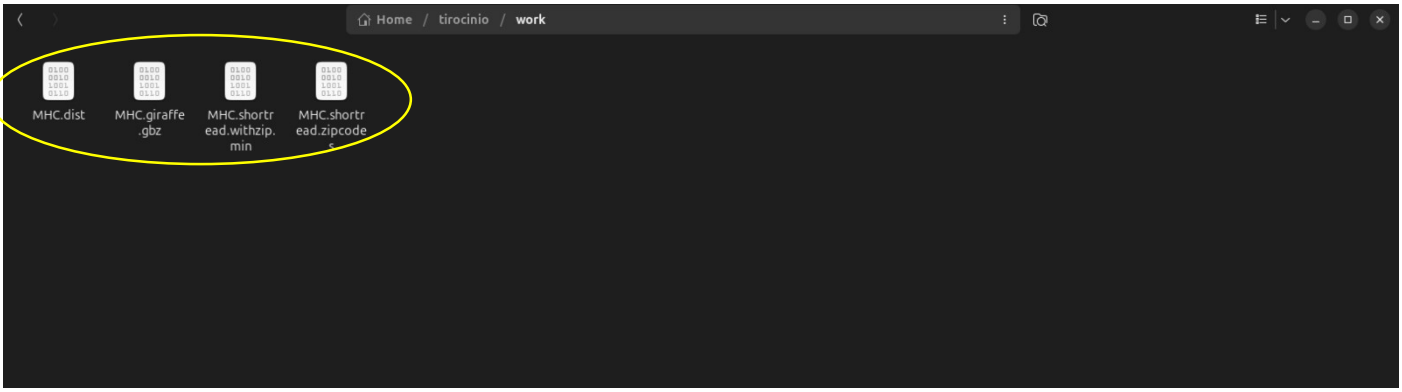
MHC.dist (~ 89.3 MB): È un indice delle distanze nel grafo.

Serve a Giraffe per capire quanto sono lontani due punti nel grafo (un concetto che nel genoma lineare è banale, ma nel grafo no).

MHC.shortread.withzip.min (~ 2.2 GB): È un indice di ricerca veloce chiamato minimizer index.

Contiene tanti piccoli pezzi di DNA (k-mer da 29 basi) che servono per trovare velocemente dove una read può attaccarsi nel grafo.

MHC.shortread.zipcodes (~ 604.5 kB): È un file accessorio al .min. Contiene informazioni extra (dette “zipcodes”) che indicano dove certi pezzetti particolari si trovano nel grafo. Serve solo a completare l’indice .min in alcuni casi.



Informazioni di tragitto: statistiche sulle varianti e cosa ci aspetta

Di seguito alcune informazioni interessanti:

Numero di **varianti nel VCF**, regione MHC

```
bcftools stats data/MHC.1000G.phased.vcf.gz > work/MHC.vcf.stats
grep ^SN work/MHC.vcf.stats | cut -f3-
```

```
number of samples:   2548    # Campioni
number of records:   150572  # Varianti totali
number of no-ALTs:   0
number of SNPs:      140790  # SNP
number of MNPs:      0
number of indels:    9782    # Indel
number of others:    0
number of multiallelic sites:      0
number of multiallelic SNP sites:  0
```

Numero di **varianti nel grafo**

```
vg deconstruct -P 6 -a -e -O work/MHC.giraffe.gbz > work/MHC.from_graph.vcf
```

```
Warning [vg deconstruct]: -e is deprecated as it's now on default
```

```
bcftools stats work/MHC.from_graph.vcf > work/MHC.graph.stats
grep ^SN work/MHC.graph.stats | cut -f3-
```

```
number of samples:   64
number of records:   51043
number of no-ALTs:   0
number of SNPs:      46474
number of MNPs:      1576
number of indels:    4020
number of others:    194
number of multiallelic sites:      1980
number of multiallelic SNP sites:  728
```

Il grafo contiene ~51k varianti. Sono meno di 150k perché il grafo accorpa varianti vicine/collegate in "bolle". Più varianti bialleliche vicine possono diventare un solo sito multiallelico/complesso nel grafo; ecco perché qui compaiono multiallelic e MNP (mentre il VCF input era tutto biallelico e MNP=0). È normale che i conteggi non coincidano 1:1.

Quante **aplotipi** ci sono **nella GBWT**

```
vg gbwt -Z work/MHC.giraffe.gbz -M -H -S -L | sed -n '1,200p'
```

```
65 paths with names, 65 samples with names, 65 haplotypes, 1 contigs with names
65
_gbwt_ref
path_cover_0
…
path_cover_63
```

Quando *vg autoindex* costruisce la GBWT, cerca di includere i cammini reali (gli aplotipi) che trova nel VCF.
Se per qualche motivo non riesce o non li importa (per esempio: ritaglio regionale, header non perfettamente allineati, ecc.), il programma non si ferma: genera da solo dei cammini "fittizi" chiamati path-cover.

- Questi *path-cover* sono 64 percorsi sintetici (per impostazione predefinita).
- Servono a coprire tutto il grafo in modo che Giraffe abbia comunque dei "sentieri plausibili" per orientarsi quando mappa.
- Non aggiungono nuove varianti: attraversano i nodi e gli archi che già esistono nel grafo.
- In pratica, è come dire:

"Non so quali siano i veri percorsi di tutti gli individui, quindi ne creo 64 che passino ovunque, così il grafo è completo e navigabile."

Ora devo testare e analizzare come funziona il mapper *Giraffe*, cioè l'algoritmo che allinea le letture di sequenziamento sul grafo. Tuttavia per fare ciò bisogna sapere che ogni lettura (read) è una piccola porzione dell'intero genoma, e viene memorizzata in formato FASTQ (Fast Quality), che contiene:

- La sequenza di basi (le lettere del DNA: A, C, G, T).
- La qualità per base (Phred score): quanto è "affidabile" ogni lettera letta.

L'input tipico di *Giraffe* sono proprio i file in formato FASTQ.

Struttura di un file *.fq*

Riga	*Contenuto*	*Descrizione*
1	@IDENTIFICATIVO	Inizia con @, contiene il nome o ID della lettura (spesso include info sullo strumento di sequenziamento).
2	SEQUENZA	La sequenza nucleotidica letta (caratteri A, C, G, T, N).
3	+	Separatore tra sequenza e qualità.
4	QUALITÀ	Una stringa di caratteri ASCII che codifica la qualità (affidabilità) di ogni base della sequenza: ogni carattere rappresenta un punteggio Phred.

Idealmente si potrebbero utilizzare **dati reali** provenienti da esperimenti di sequenziamento, oppure utilizzare **letture simulate**.

La seconda opzione è preferibile per diversi motivi, tra cui:

1. Scarsa reperibilità di FASTQ reali adatti allo scopo
2. Dimensione e Costo Computazionale
3. Più problemi da gestire
4. Semplicità di valutazione dell'accuratezza dell'allineamento: la simulazione permette di conoscere con esattezza la sequenza di origine di ciascuna lettura, con dati reali invece, non posso sapere con certezza se una lettura "sbagliata" dipende da un errore della macchina che l'ha sequenziata, da una mutazione vera o da un errore di Giraffe.

Tuttavia tale opzione ha anche dei lati negativi:

1. Considerevole percentuale di irrealismo biologico
2. Valutazioni parziali delle prestazioni

Per completezza, le proverò entrambe, prima però:

Problema 6: Necessità dell'indice XG

Molti comandi del pacchetto vg, che utilizzerò nelle fasi successive alla costruzione del grafo, non leggono il formato GBZ, bensì richiedono l'indice XG.

Per questo motivo, pur mantenendo il file *.gbz* (che verrà utilizzato da *Giraffe* per la mappatura), è necessario convertirlo in formato *.xg*.

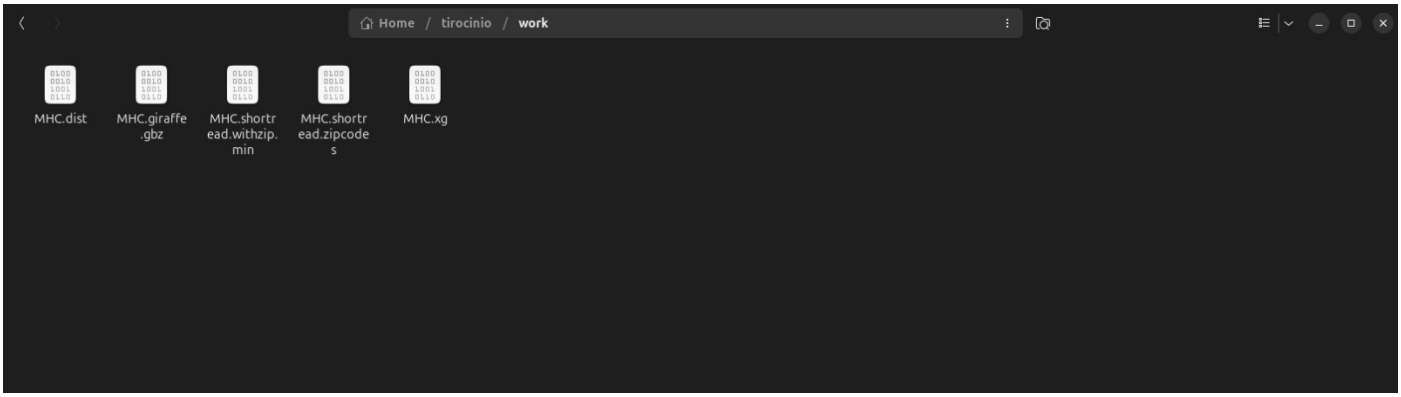
L'indice *.xg* è indispensabile per numerosi comandi a valle, tra cui:

- vg sim, per la simulazione di letture;
- vg surject, per la conversione degli allineamenti in coordinate lineari;
- vg stats, per il calcolo di statistiche sugli allineamenti;
- e alcuni comandi di *vg gbwt*.

Conversione del grafico in formato XG

```
vg convert -x --drop-haplotypes work/MHC.giraffe.gbz > work/MHC.xg
```

~ 301.9 MB



Problema 6.1: Opzione `--drop-haplotype`

Durante la conversione da formato GBZ a XG, si utilizza l'opzione `--drop-haplotypes`, per far sì che l'indice XG mantenga solo il percorso di riferimento principale (nel nostro caso, il cromosoma 6) ed escluda i percorsi aplotipici contenuti nella GBWT. Questa scelta è fondamentale per motivi sia tecnici che logici, elencati di seguito.

- Efficienza: evita che l'XG diventi enorme (decine di GB) e instabile, mantenendolo leggero e veloce da caricare.
- Proiezione coerente (surjection): semplifica la conversione degli allineamenti su un'unica linea di riferimento, evitando ambiguità dovute a centinaia di percorsi alternativi.
- Compatibilità: i file BAM a seguire, risulteranno piccoli e facilmente gestibili; avranno coordinate e statistiche coerenti con strumenti standard (*samtools*, *bcftools, …*) e manterranno la coerenza dei nomi tra FASTA, VCF e grafo.
- Nessun impatto biologico: il mapping non cambierà, perché *Giraffe* usa il GBZ (che contiene tutti gli aplotipi), mentre l'XG serve solo per analisi successive.

Preparazione della directory per le reads simulate, per il mapping, per le statistiche e per i log

```
mkdir -p reads maps stats logs
```

Adesso siamo pronti per gli esperimenti:

- *(Operazione preliminare obbligatoria)* Costruzione degli indici per le due configurazioni, cliccare **qui**.
- Per l'esperimento su dati simulati, cliccare **qui**.
- Per l'esperimento su dati reali, cliccare **qui**.

Continuando di seguito invece sarà possibile visualizzare l'esecuzione di un Sanity Check.

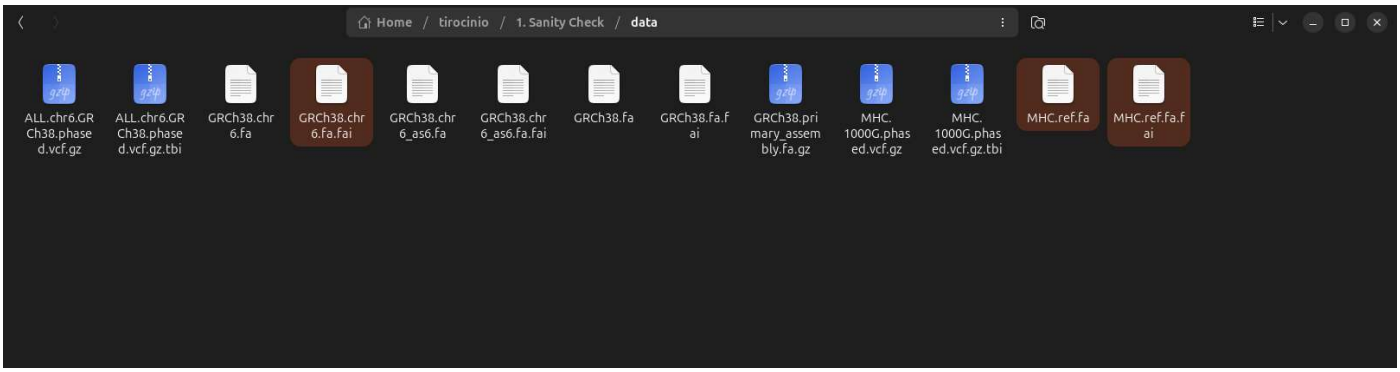
Sanity Check

In questa fase voglio verificare che la pipeline di allineamento basata su *vg giraffe* funzioni correttamente, prima di passare agli esperimenti veri e propri. Per farlo simulerò letture direttamente dal riferimento (quindi perfettamente coerenti con il grafo), e mi aspetterò un allineamento corretto e senza errori.

Estrazione della regione MHC dal cromosoma 6

```
samtools faidx data/GRCh38.chr6.fa chr6:28477797-33448354 > data/MHC.ref.tmp.fa
( echo ">6"; tail -n +2 data/MHC.ref.tmp.fa ) > data/MHC.ref.fa
rm -f data/MHC.ref.tmp.fa
samtools faidx data/MHC.ref.fa
```

GRCh38.chr6.fa.fai (~ 23 bytes)
MHC.ref.fa (~ 5.1 MB)
MHC.ref.fa.fai (~ 18 bytes)



Simulazione di letture di sequenziamento a partire dal grafo genomico

Problema 7: il limite di *vg sim*

Verrebbe in mente di utilizzare il comando *vg sim*, appositamente progettato per simulare letture; tale comando però genera in output un file in formato GAM e non FASTQ, questo perché i dati sono simulati, ed è noto fin dall'inizio dove si trovano le letture nel genoma di riferimento, quindi è come se fossero anche già state allineate (infatti dopo l'allineamento si ottiene proprio un file in formato GAM), questa cosa si chiama ground truth: è una scelta progettuale che però impone un limite particolare, cioè l'impossibilità di testare l'algoritmo di allineamento. Questo non ci permette di studiare gli effetti della GBWT.

Si può quindi pensare di ricavare dal GAM di *vg sim* i **due** file FASTQ per poi effettuare manualmente l'allineamento. Per estrarre i FASTQ l'unica opzione è utilizzare il comando *vg view* (con opzione -X per le forward reads (R1) e -Y per le reverse reads (R2)) sul file GAM.

Perché vogliamo **due** FASTQ? Nel sequenziamento paired-end (doppia terminazione), ogni frammento di DNA viene sequenziato da entrambe le estremità, generando file R1 (forward) e R2 (reverse) accoppiati. Serve ad avere un allineamento ottimale: se una delle due letture cade in una regione ambigua, l'altra può "ancorarla" nel punto giusto.

Qui però sorge il:

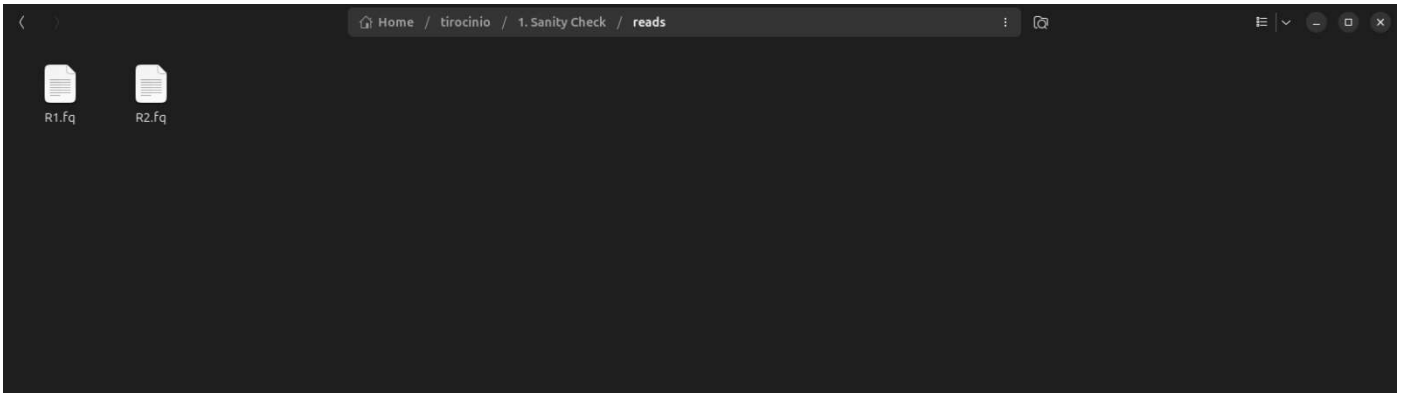
Problema 8: R1 ≠ R2 dall'estrazione GAM

Il file GAM contiene coppie valide (`vg view -aj reads/file.gam | head`), infatti il problema risiede nell'estrazione (non nella simulazione): purtroppo l'opzione `-X/-Y` può dare risultati inattesi su alcune versioni di *vg*, a causa di bug.
Soluzione più semplice e adatta allo scopo:
Generare i FASTQ con *wgsim* (simulatore di letture progettato per operare su un genoma di riferimento lineare).

```
wgsim -N 100000 \
      -1 150 -2 150 \
      -e 0.001 \
      -r 0 \
      -R 0 \
      -d 400 -s 50 \
      data/MHC.ref.fa reads/R1.fq reads/R2.fq
```

`-N 100000:`	numero coppie di letture: 200.000 totali (R1 + R2)
`-1 150 -2 150:`	lunghezza di ciascuna lettura (150 basi)
`-e 0.001:`	% di errori casuali nelle basi
`-r 0:`	% di mutazione reale introdotta
`-R 0:`	% di indel
`-d 400 -s 50:`	lunghezza media dei frammenti di 400 basi ± 50
`data/GRCh38.chr6_as6.fa:`	sequenza da cui simulare
`reads/R1.fq, reads/R2.fq:`	output FASTQ paired-end

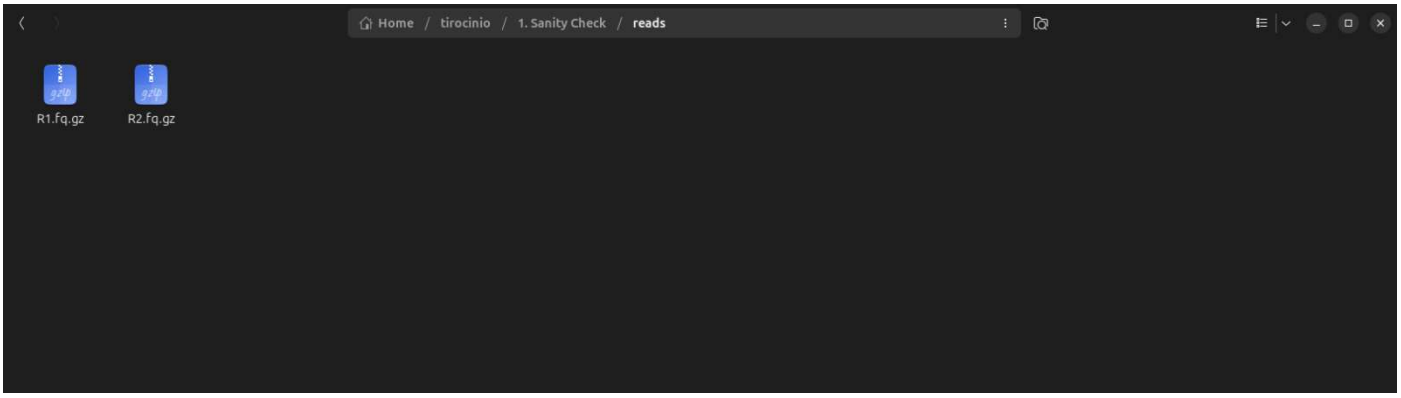
~ 34.2 MB, ~ 34.2 MB



Compressione

```
gzip reads/R1.fq reads/R2.fq
```

~ 6.3 MB, ~ 6.3 MB



Verifica

```
zcat reads/R1.fq.gz | wc -l
```

Output: 400.000 righe → 100.000 reads

```
zcat reads/R2.fq.gz | wc -l
```

Output: 400.000 righe → 100.000 reads

OK.

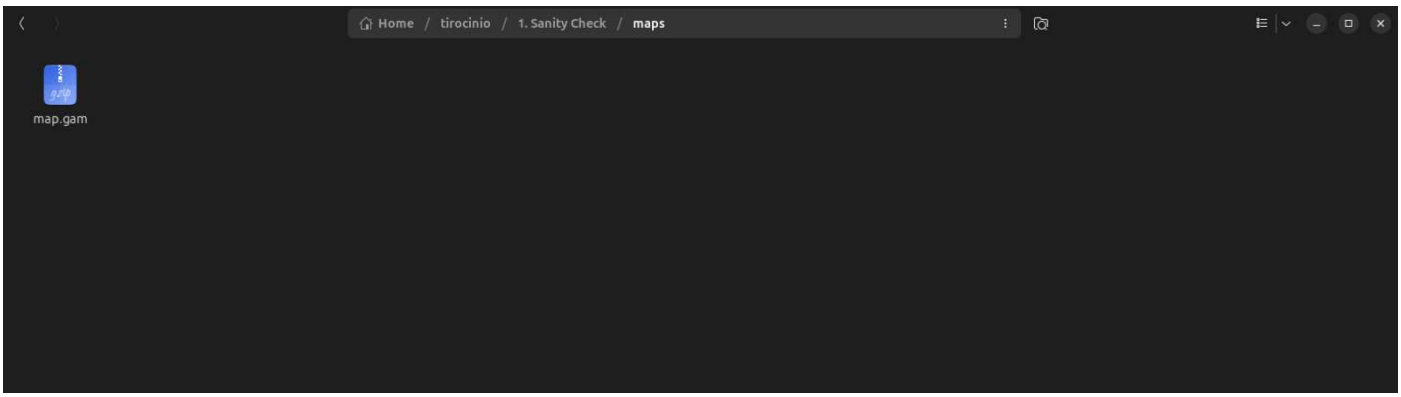
Allineamento con Giraffe (configurazione standard: *path-cover* = 64)

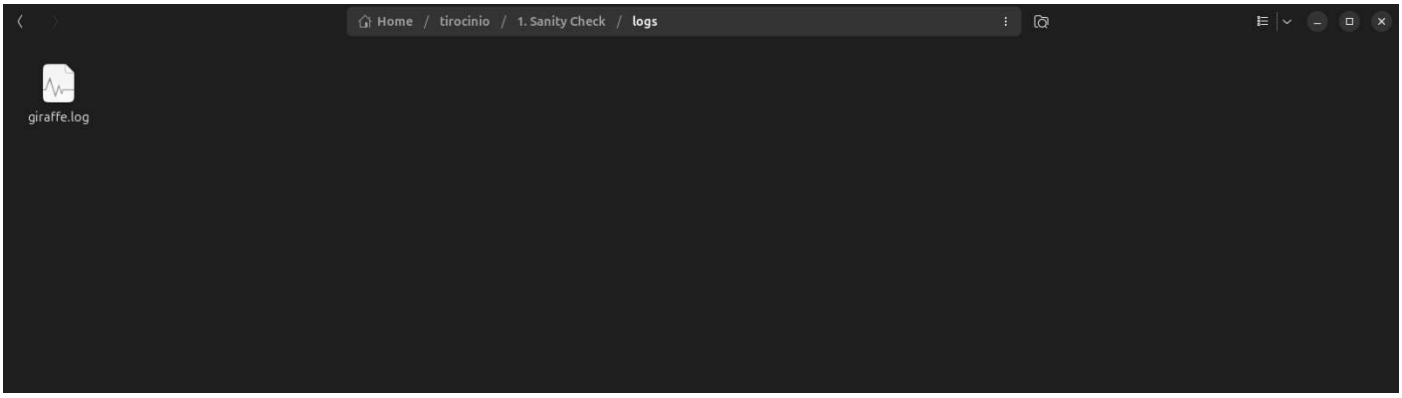
```
(/usr/bin/time -v vg giraffe \
  -Z work/MHC.giraffe.gbz \
  -m work/MHC.shortread.withzip.min \
  -d work/MHC.dist \
  -f reads/R1.fq.gz \
  -f reads/R2.fq.gz \
  -t 16 -p > maps/map.gam) 2> logs/giraffe.log
```

`-p`: output leggibile

`2> giraffe.A.log`: salva il log con tempi e RAM

~ 40.3 MB, ~ 4 kB





Problema 9: Il formato GAM e la conversione in BAM

Dopo l'allineamento, *vg giraffe* produce un file di output in formato GAM.
Questo formato conserva informazioni sugli allineamenti nel grafo, ma non è leggibile dagli strumenti standard (come *samtools* o *bcftools*), che operano invece su coordinate lineari (BAM/SAM).
Per poter analizzare e visualizzare i risultati con strumenti comuni, occorre quindi convertire il file *.gam* in *.bam* su coordinate del riferimento lineare.
Tale conversione viene effettuata tramite il comando *vg surject*.

Problema 9.1: Perdita di informazione

L'operazione di surjection comporta una proiezione dell'allineamento sul percorso di riferimento del grafo.
In particolare, vg surject:

1. individua, per ogni lettura, il percorso di riferimento più vicino;
2. proietta la posizione dell'allineamento lungo tale path;
3. genera un CIGAR che descrive come la read si allinea rispetto alla sequenza di riferimento.

Con questa operazione si mantiene:

- la posizione lineare della read sul riferimento;
- le informazioni di base per il calcolo di identità, coverage ed error rate;
- la possibilità di individuare regioni variabili come mismatch o indel.

Tuttavia, si perde la componente topologica del grafo:

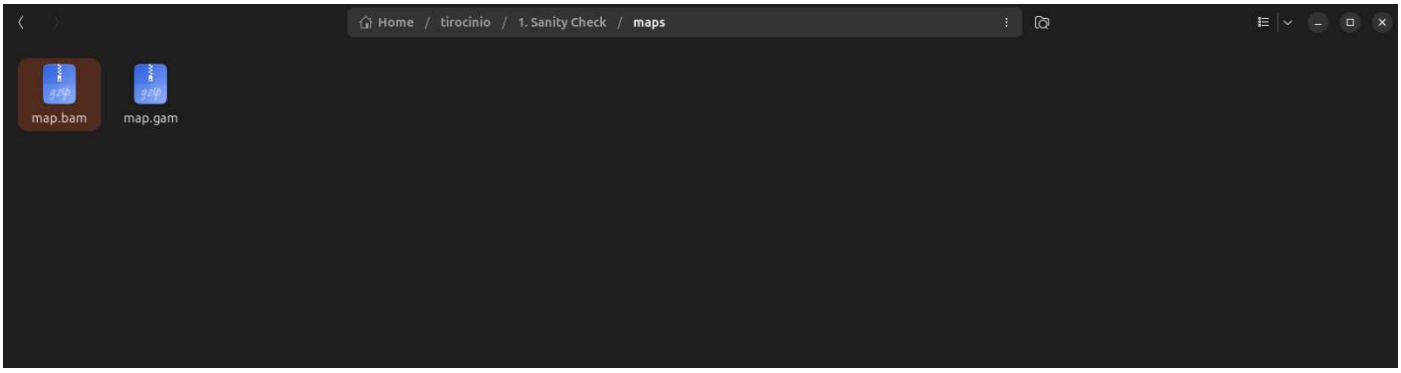
- non è più possibile sapere su quale ramo alternativo (allele) la read era allineata;
- eventuali letture su varianti aplotipiche vengono "schiacciate" sul riferimento lineare come errori o indel.

Sebbene ciò comporti una perdita di dettaglio, non rappresenta un problema per l'obiettivo del progetto, che è quello di valutare quanto l'uso del GBWT migliori il mapping.
Le informazioni topologiche perse nella surjection non influiscono sui parametri di qualità del mapping, che vengono invece calcolati in fase di allineamento.

Surject: "proietto" gli allineamenti del grafo GAM su un genoma lineare BAM, questo solo per poter analizzare i risultati con *samtools*, i quali possono essere usati per un confronto con altri mappatori, ad es. *BWA*.

```
vg surject -p 6 -x work/MHC.xg -b maps/map.gam | samtools sort -o maps/map.bam
```

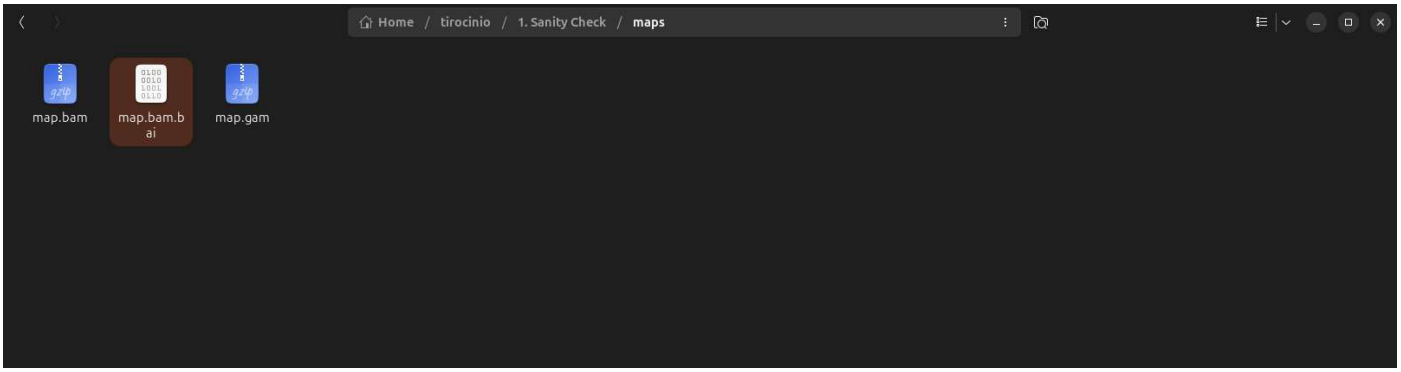
~ 7.2 MB



Indicizzazione del file BAM: viene generato l'indice .bai associato al file BAM, necessario per consentire accesso rapido alle regioni e calcolo delle statistiche.

```
samtools index maps/map.bam
```

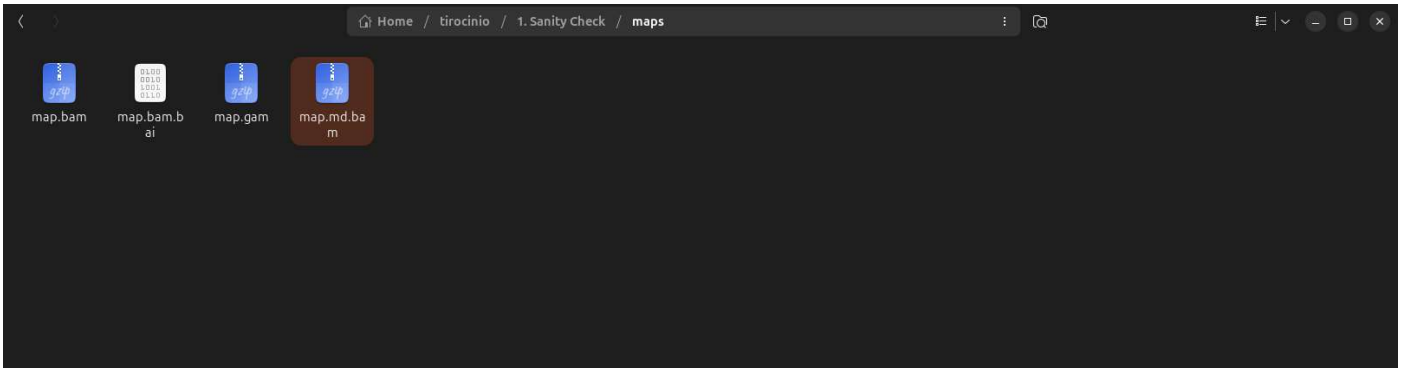
~ 73.9 kB



Aggiunta di tag MD/NM: informazioni di mismatch (MD) e distanza di editing (NM), richieste da samtools stats.

```
samtools calmd -b maps/map.bam data/MHC.ref.fa > maps/map.md.bam
```

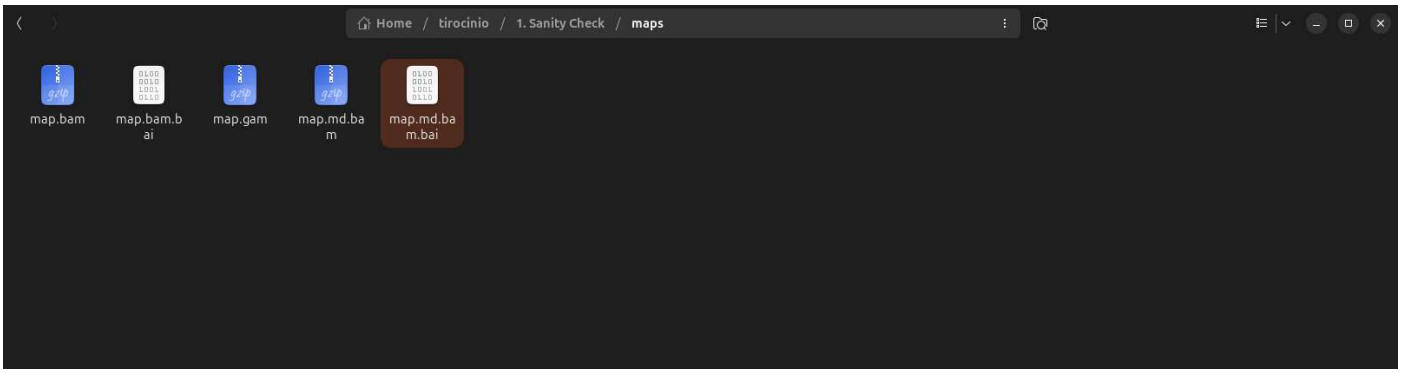
~ 7.2 MB



Indicizzazione

```
samtools index maps/map.md.bam
```

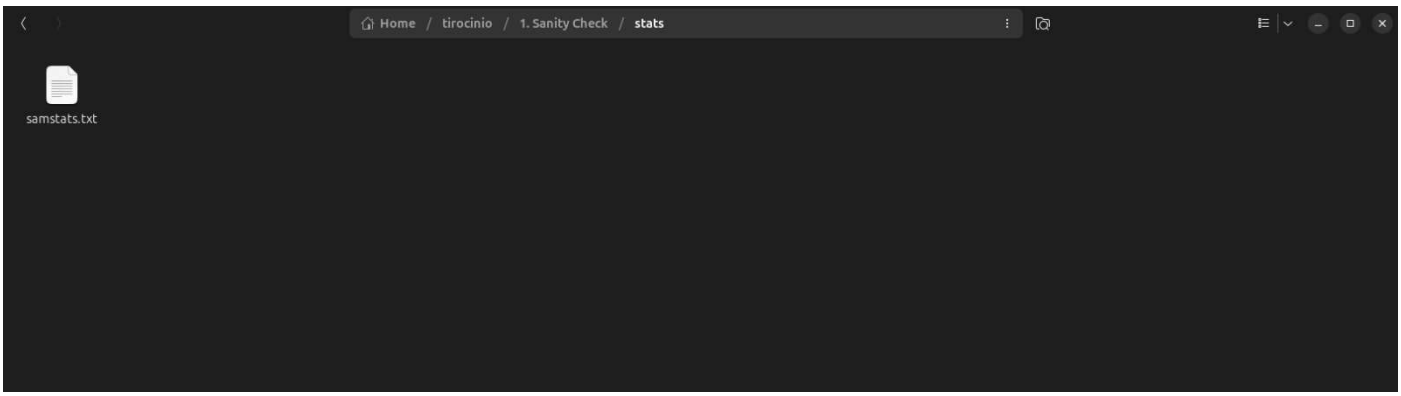
~ 73.9 kB



Calcolo statistiche generali

```
samtools stats maps/map.md.bam > stats/samstats.txt
```

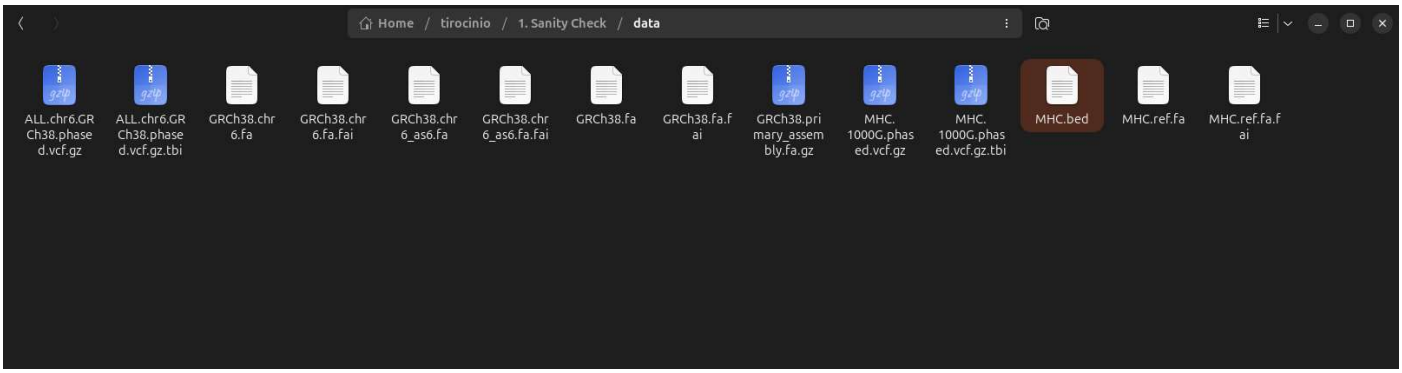
~ 37.9 kB



Definisco la regione sulla quale *samtools* dovrà lavorare

```
echo -e "6\t28477797\t33448354" > data/MHC.bed
```

~ 20 bytes



Numero di letture primarie allineate nella MHC e MQ0 fraction (valore di ambiguità): mi aspetto:
~200 000 letture e MQ0_fraction ≈ 0

```
samtools view -F 2308 -L data/MHC.bed maps/map.bam | \
  awk '{mq=$5; n++; if(mq==0) z++} END{
    printf("Primari in MHC: n=%d, MQ0_fraction=%.4f\n", (n? n:0), (n? z/n:0))
  }'
```

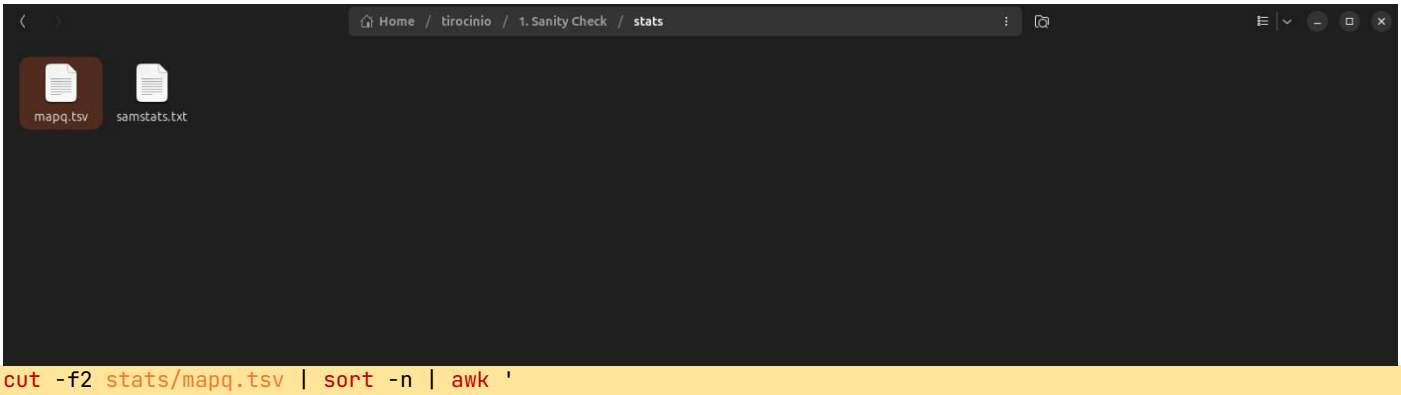
```
Primari in MHC: n=199990, MQ0_fraction=0.0001
```

Ottimo.

Analisi della distribuzione di MAPQ (valori di qualità): mi aspetto: p50 = 60, p90 = 60

```
vg view -aj maps/map.gam | jq -r '[.name, .mapping_quality, .score] | @tsv' > stats/mapq.tsv
```

~ 8.8 MB




```
cut -f2 stats/mapq.tsv | sort -n | awk '
  {a[NR]=$1}
  END{
    n=NR
    if(n==0){print "MAPQ: nessuna read"; exit}
    p50=a[int((n+1)/2)]
    p90=a[int(0.9*n)]
    printf("MAPQ: p50=%.1f, p90=%.1f, n=%d\n", p50, p90, n)
  }'
```

```
MAPQ: p50=60.0, p90=60.0, n=200000
```

Ottimo.

Tempo e memoria

```
grep -E "Elapsed \(wall clock\)|Maximum resident set size" logs/giraffe.log
```

```
Elapsed (wall clock) time (h:mm:ss or m:ss): 0:14.92
Maximum resident set size (kbytes): 3024440
```

la pipeline è validata: si può procedere in sicurezza con gli esperimenti successivi.

Fase 3: “Costruzione degli indici per le altre due configurazioni (path-cover: n=16 ed n=128”

Questa fase è necessaria sia per il mapping sintetico (Fase 4) che per il mapping reale (Fase 5).

Problema 10: `gbwt` è difficile da usare

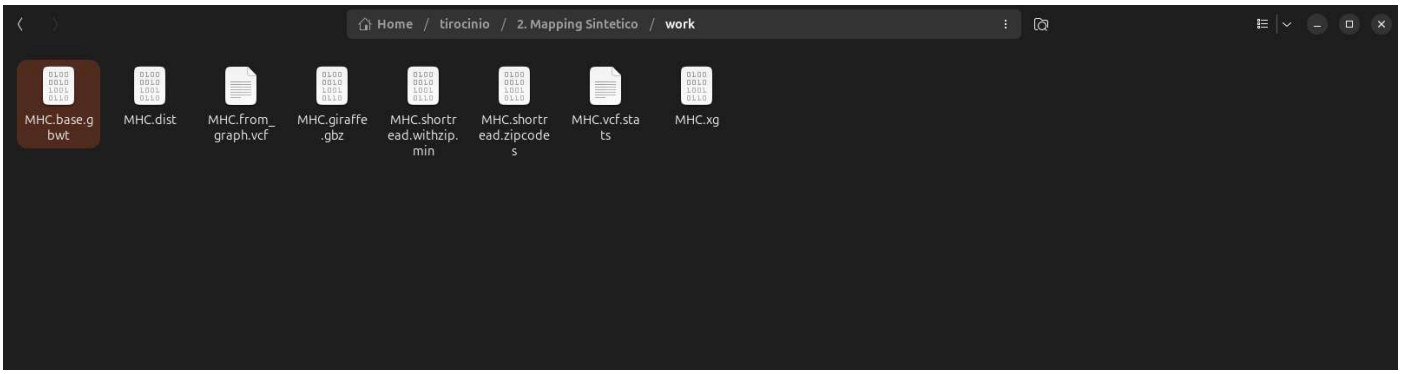
È sconsigliato nei workflow pangenomici con molti campioni, perché la pipeline è complessa, richiede input molto curati (path, formati, compatibilità) ed è più soggetta a errori. Le versioni recenti di *vg* raccomandano di lavorare direttamente con il formato GBZ, più stabile e integrato.

```
cd work
```

Estrazione della GBWT standard dal GBZ

```
vg gbwt -Z MHC.giraffe.gbz -o MHC.base.gbwt
```

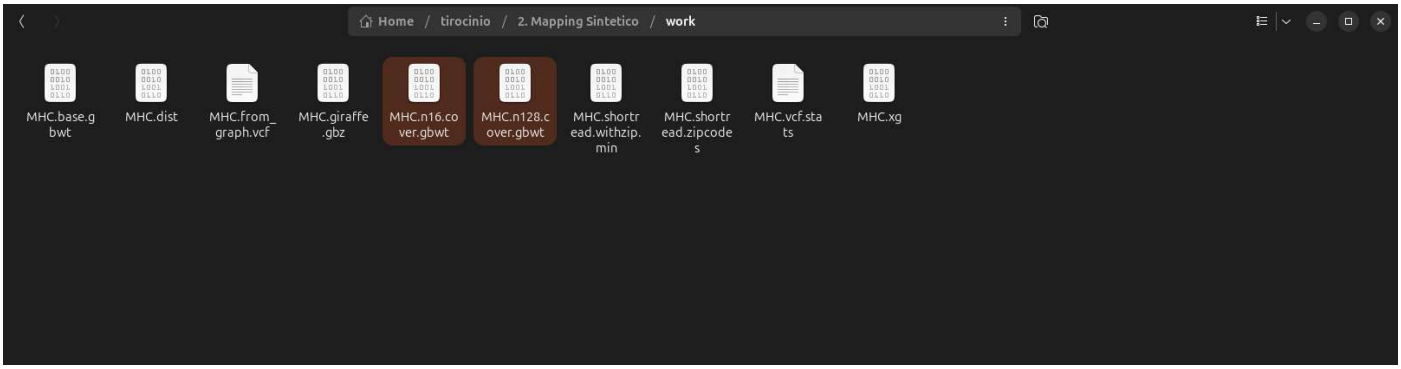
~ 87.8 MB



path-cover per n=16 ed n=128 (lo standard è quello estratto)

```
vg gbwt -x MHC.xg -o MHC.n16.cover.gbwt  -P -n 16
vg gbwt -x MHC.xg -o MHC.n128.cover.gbwt -P -n 128
```

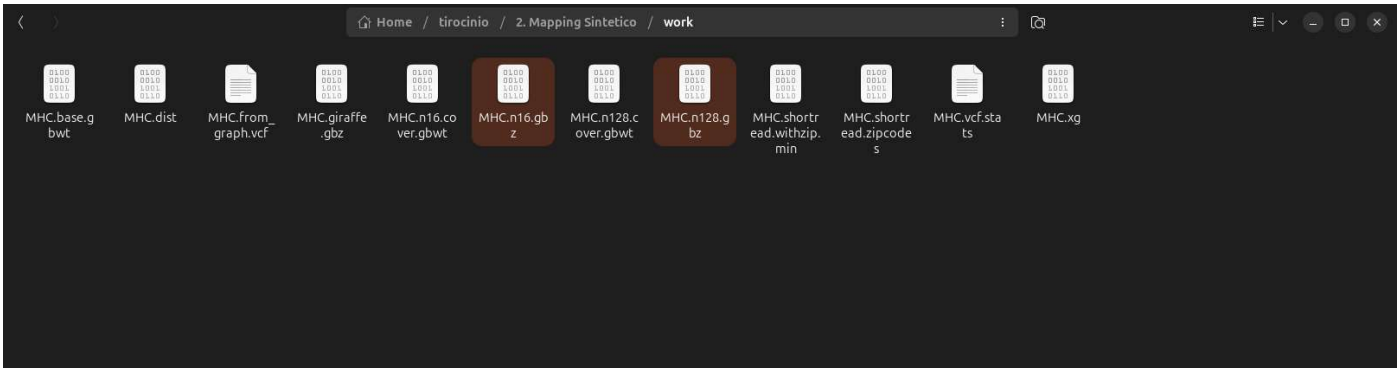
MHC.n16.cover.gbwt (~ 85.6 MB)
MHC.n128.cover.gbwt (~ 88.5 MB)



Creazione dei GBZ corrispondenti

```
vg gbwt -x MHC.xg --gbz-format -g MHC.n16.gbz  MHC.n16.cover.gbwt
vg gbwt -x MHC.xg --gbz-format -g MHC.n128.gbz MHC.n128.cover.gbwt
```

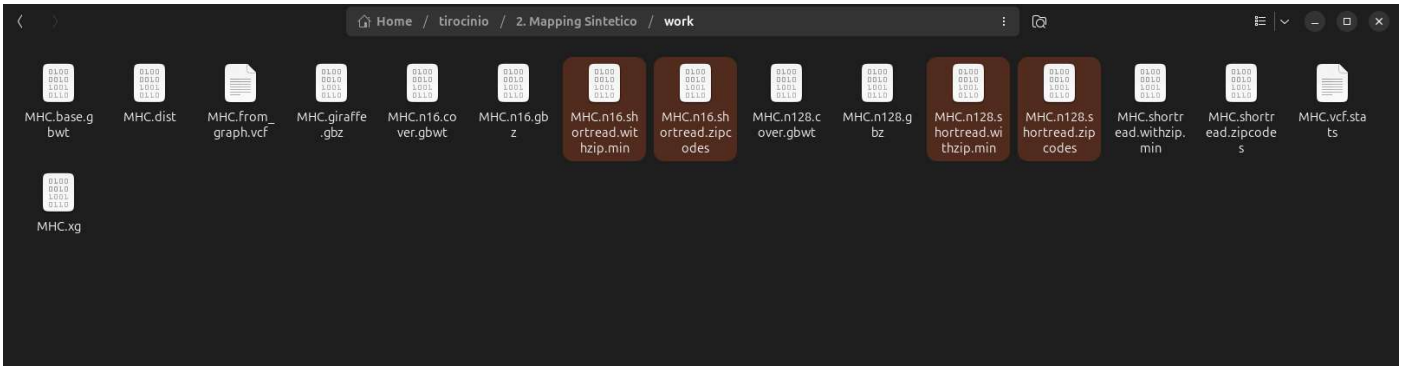
MHC.n16.gbz (~ 154.6 MB)
MHC.n128.gbz (~ 157.5 MB)



Creazione dei minimizer per ciascun GBZ

```
vg minimizer -t 16 -d MHC.dist -o MHC.n16.shortread.withzip.min -z MHC.n16.shortread.zipcodes MHC.n16.gbz
vg minimizer -t 16 -d MHC.dist -o MHC.n128.shortread.withzip.min -z MHC.n128.shortread.zipcodes MHC.n128.gbz
```

MHC.n16.shortread.withzip.min (~ 2.2 GB)
MHC.n16.shortread.zipcodes (~ 642.5 kB)
MHC.n128.shortread.withzip.min (~ 2.2 GB)
MHC.n128.shortread.zipcodes (~ 644.1 kB)



Fase 4: “Allineamento di letture haplotype-aware (simulate)”

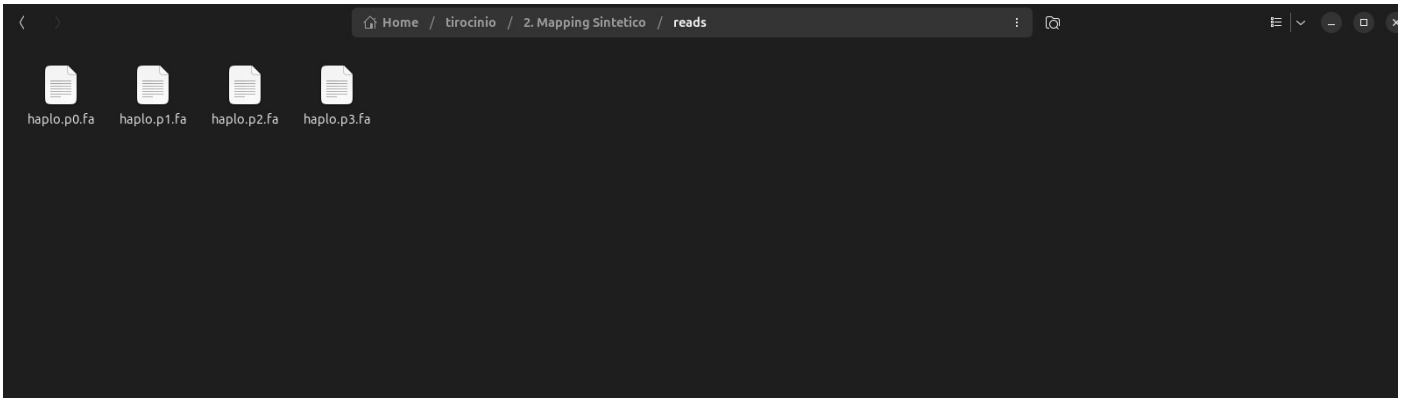
In questa fase voglio verificare se la densità dell’indice aplotipico (GBWT con n=16, n=64, n=128) influisce sulla qualità e sull’efficienza del mapping con vg giraffe.
Le letture verranno simulate da 4 path-cover distinti, per un totale di 100.000 coppie (paired-end 150 bp, errore 1%).

Problema 11: come simulare letture “haplotype-aware”?
Constatato il limite di *vg sim* e il fatto che wgsim simula letture a partire da riferimenti lineari, allora come simulo letture che tengono conto degli aplotipi?
L’escamotage è usare il vg toolkit per estrarre le sequenze lineari degli aplotipi desiderati, e successivamente usare wgsim per generare i file FASTQ da queste sequenze lineari

Estrazione FASTA dagli aplotipi scelti

```
vg paths -x MHC.xg -g MHC.base.gbwt -Q path_cover_0#0#6#0 -F > ../reads/haplo.p0.fa
vg paths -x MHC.xg -g MHC.base.gbwt -Q path_cover_1#0#6#0 -F > ../reads/haplo.p1.fa
vg paths -x MHC.xg -g MHC.base.gbwt -Q path_cover_2#0#6#0 -F > ../reads/haplo.p2.fa
vg paths -x MHC.xg -g MHC.base.gbwt -Q path_cover_3#0#6#0 -F > ../reads/haplo.p3.fa
```

haplo.p0.fa (~ 172.9 MB)
haplo.p1.fa (~ 172.9 MB)
haplo.p2.fa (~ 172.9 MB)
haplo.p3.fa (~ 172.9 MB)

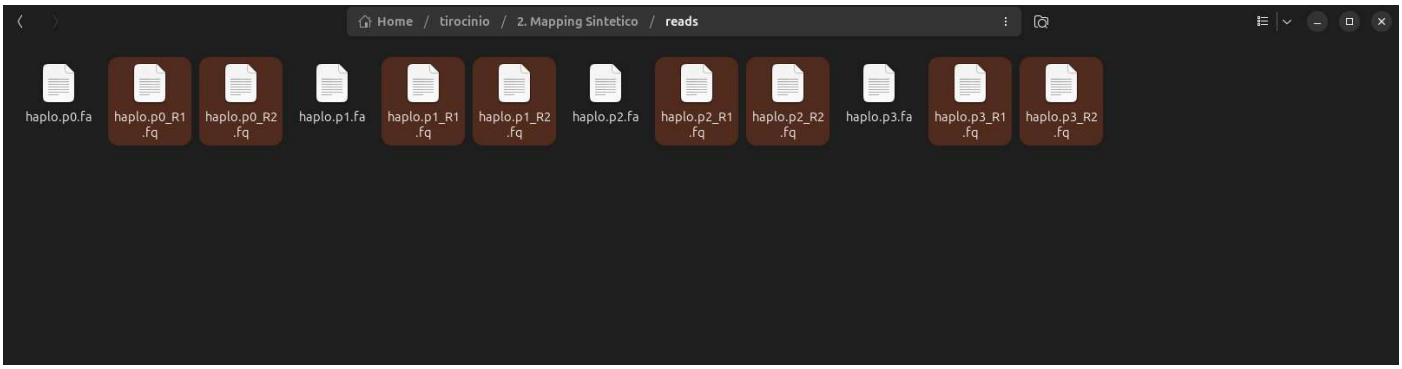



```
cd ../reads
```

Simulo le letture

```
wgsim -N 25000 -1 150 -2 150 -e 0.01 -r 0 -R 0.001 -d 400 -s 50 haplo.p0.fa haplo.p0_R1.fq haplo.p0_R2.fq
wgsim -N 25000 -1 150 -2 150 -e 0.01 -r 0 -R 0.001 -d 400 -s 50 haplo.p1.fa haplo.p1_R1.fq haplo.p1_R2.fq
wgsim -N 25000 -1 150 -2 150 -e 0.01 -r 0 -R 0.001 -d 400 -s 50 haplo.p2.fa haplo.p2_R1.fq haplo.p2_R2.fq
wgsim -N 25000 -1 150 -2 150 -e 0.01 -r 0 -R 0.001 -d 400 -s 50 haplo.p3.fa haplo.p3_R1.fq haplo.p3_R2.fq
```

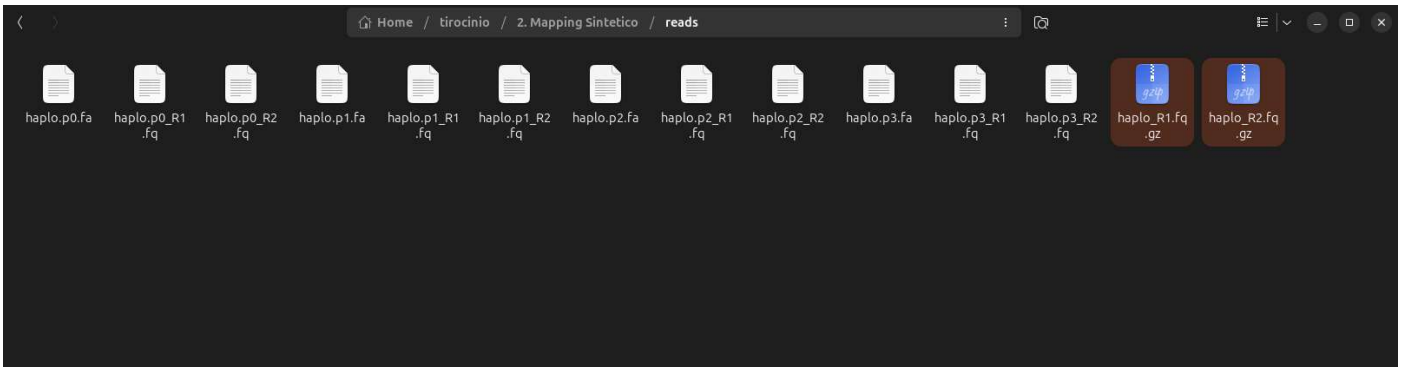
haplo.p0_R1.fa (~ 9 MB)
haplo.p0_R2.fa (~ 9 MB)
haplo.p1_R1.fa (~ 9 MB)
haplo.p1_R2.fa (~ 9 MB)
haplo.p2_R1.fa (~ 9 MB)
haplo.p2_R2.fa (~ 9 MB)
haplo.p3_R1.fa (~ 9 MB)
haplo.p3_R2.fa (~ 9 MB)



Unione e compressione in un unico set R1/R2

```
cat haplo.p*_R1.fq > haplo_R1.fq
cat haplo.p*_R2.fq > haplo_R2.fq
gzip -f haplo_R1.fq haplo_R2.fq
```

haplo_R1.fq.gz (~ 6.6 MB)
haplo_R2.fq.gz (~ 6.6 MB)



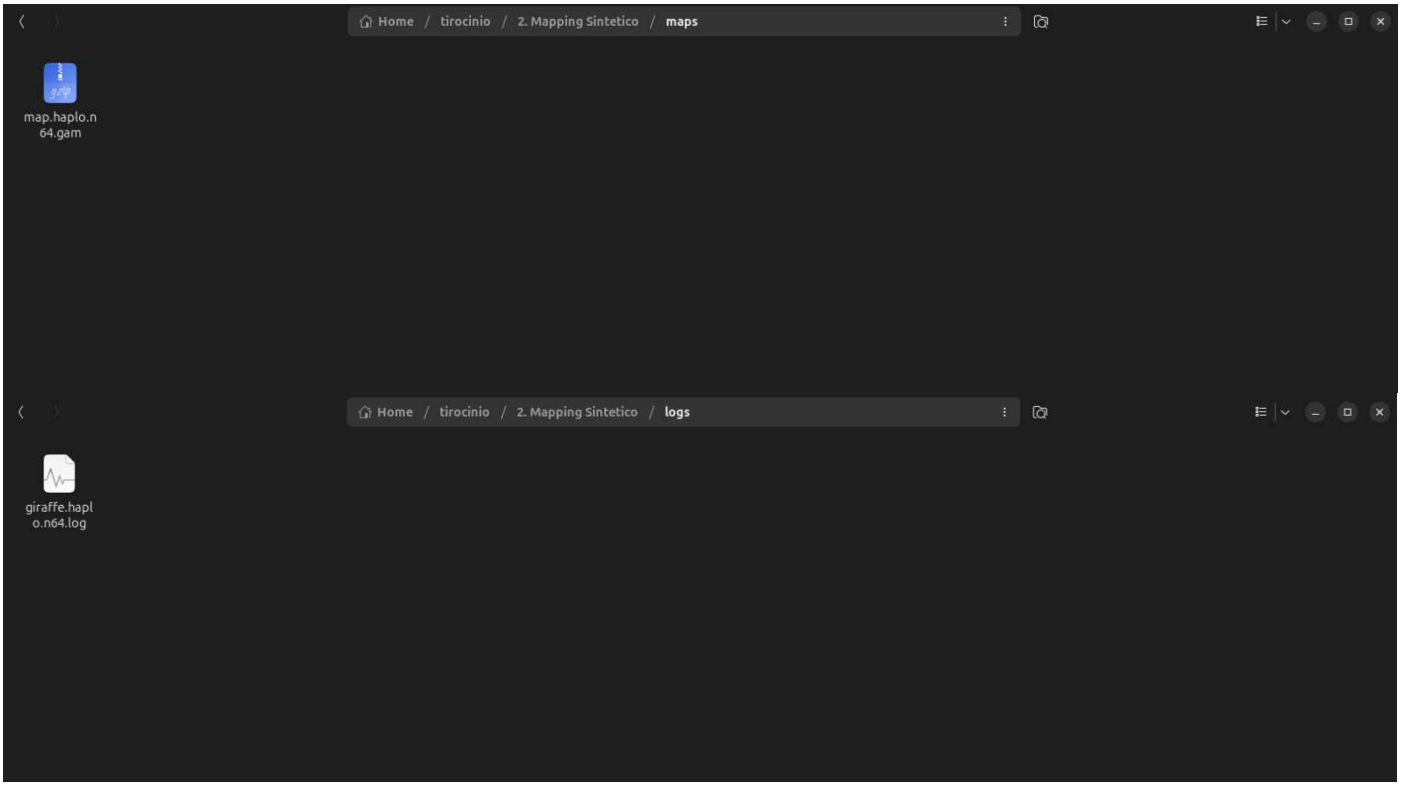

```
cd ..
```

Mapping con Giraffe nelle tre configurazioni

standard (n=64)

```
(/usr/bin/time -v vg giraffe \
  -Z work/MHC.giraffe.gbz \
  -m work/MHC.shortread.withzip.min \
  -d work/MHC.dist \
  -f reads/haplo_R1.fq.gz \
  -f reads/haplo_R2.fq.gz \
  -t 16 -p > maps/map.haplo.n64.gam) 2> logs/giraffe.haplo.n64.log
```

map.haplo.n64.gam (~ 34.9 MB)
giraffe.haplo.n64.log (~ 4 kB)

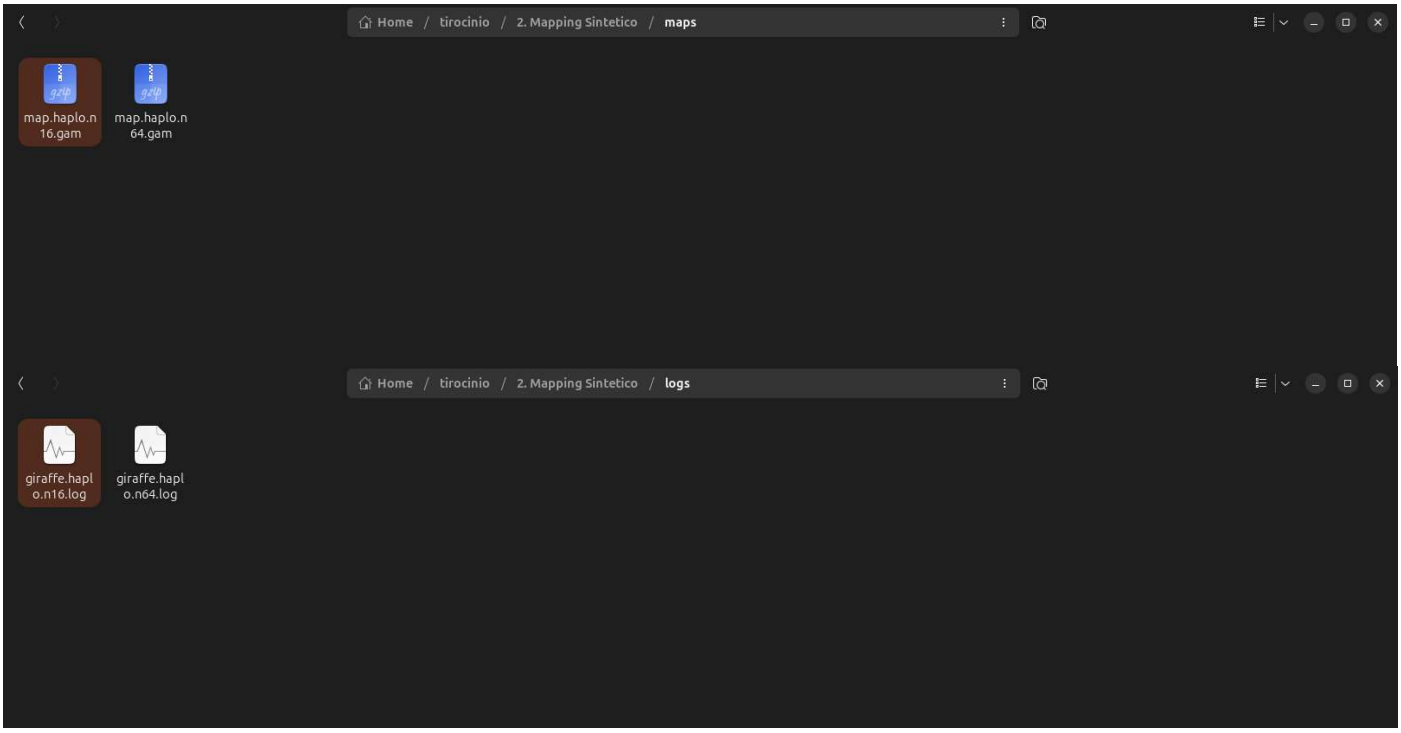


ridotta (n=16)

```
(/usr/bin/time -v vg giraffe \
  -Z work/MHC.n16.gbz \
  -m work/MHC.n16.shortread.withzip.min \
  -d work/MHC.dist \
  -f reads/haplo_R1.fq.gz \
```

```
  -f reads/haplo_R2.fq.gz \
  -t 16 -p > maps/map.haplo.n16.gam) 2> logs/giraffe.haplo.n16.log
```

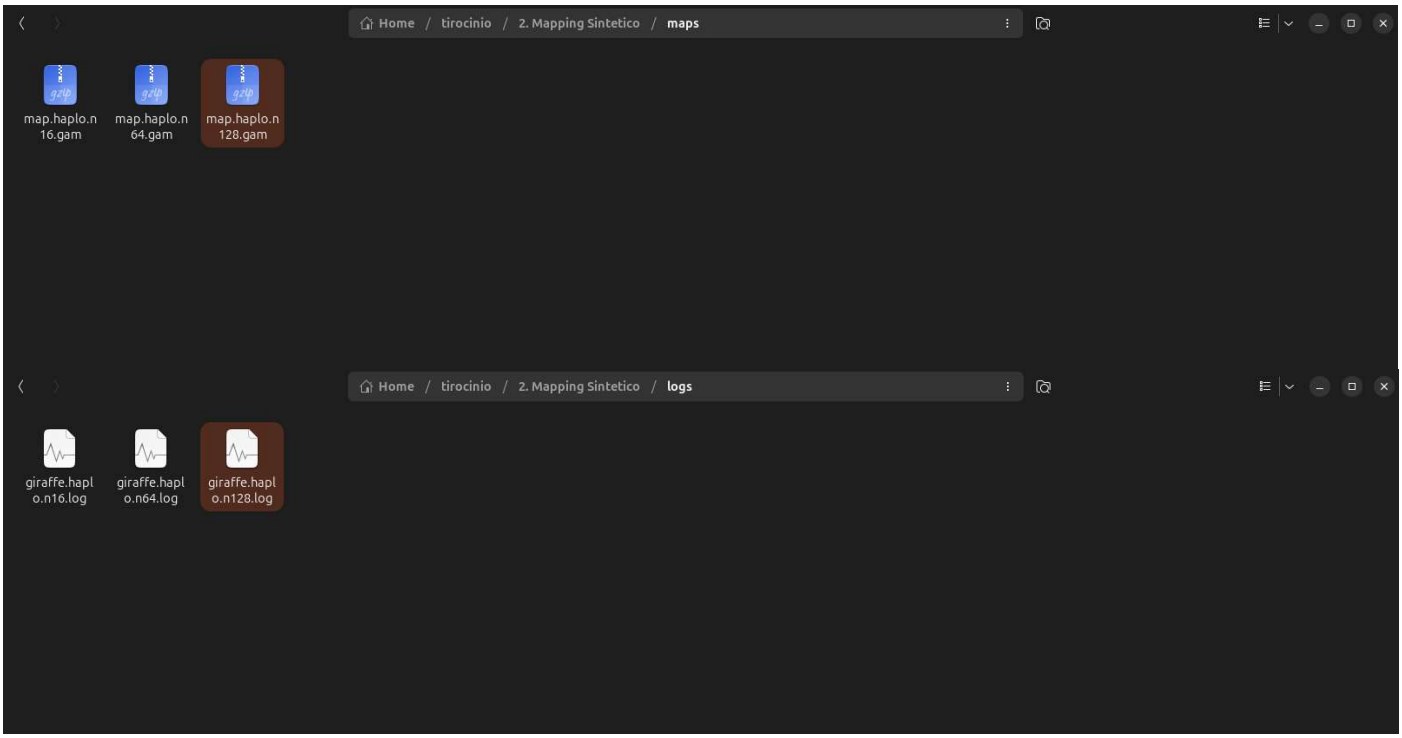
map.haplo.n16.gam (~ 34.9 MB)
giraffe.haplo.n16.log (~ 4 kB)



aumentata (n=128)

```
(/usr/bin/time -v vg giraffe \
  -Z work/MHC.n128.gbz \
  -m work/MHC.n128.shortread.withzip.min \
  -d work/MHC.dist \
  -f reads/haplo_R1.fq.gz \
  -f reads/haplo_R2.fq.gz \
  -t 16 -p > maps/map.haplo.n128.gam) 2> logs/giraffe.haplo.n128.log
```

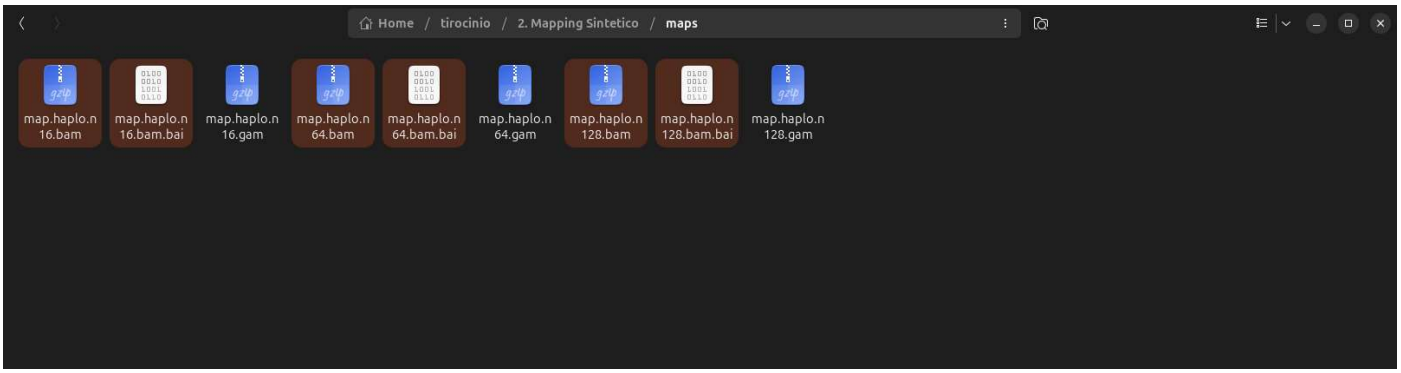
map.haplo.n128.gam (~ 34.9 MB)
giraffe.haplo.n128.log (~ 4 kB)



Surject e indicizzazione dei file BAM

```
vg surject -p 6 -x work/MHC.xg -b maps/map.haplo.n64.gam  | samtools sort -o maps/map.haplo.n64.bam
samtools index maps/map.haplo.n64.bam
vg surject -p 6 -x work/MHC.xg -b maps/map.haplo.n16.gam  | samtools sort -o maps/map.haplo.n16.bam
samtools index maps/map.haplo.n16.bam
vg surject -p 6 -x work/MHC.xg -b maps/map.haplo.n128.gam | samtools sort -o maps/map.haplo.n128.bam
samtools index maps/map.haplo.n128.bam
```

map.haplo.n16.bam (~ 12.6 MB), map.haplo.n16.bam.bai (~ 165.9 kB)
map.haplo.n64.bam (~ 12.6 MB), map.haplo.n64.bam.bai (~ 165.9 kB)
map.haplo.n128.bam (~ 12.6 MB), map.haplo.n128.bam.bai (~ 165.9 kB)



Definisco la regione sulla quale *samtools* dovrà lavorare

```
echo -e "6\t28477797\t33448354" > data/MHC.bed
```

~ 20 bytes

MQ0 fraction (ambiguità dell'allineamento)

```
for B in maps/map.haplo.n64.bam maps/map.haplo.n16.bam maps/map.haplo.n128.bam; do
  echo -n "$(basename $B) -> "
  samtools view -F 2308 -L data/MHC.bed "$B" | \
    awk '{n++; if($5==0) z++} END{printf("MQ0_fraction=%.4f (n=%d)\n",(n?z/n:0),n)}'
done
```

```
map.haplo.n64.bam -> MQ0_fraction=0.0004 (n=5650)
map.haplo.n16.bam -> MQ0_fraction=0.0004 (n=5650)
map.haplo.n128.bam -> MQ0_fraction=0.0004 (n=5650)
```

Praticamente nessuna lettura ambigua, il numero di letture allineate è costante e il risultato indica che Giraffe riesce a chiarire le letture anche con pochi path-cover.

MAPQ distribution (qualità dell'allineamento)

```
for G in maps/map.haplo.n64.gam maps/map.haplo.n16.gam maps/map.haplo.n128.gam; do
  vg view -aj "$G" | jq -r '[.name, .mapping_quality, .score] | @tsv' > "stats/mapq.$(basename $G).tsv"
  echo -n "$(basename $G) -> "
  cut -f2 "stats/mapq.$(basename $G).tsv" | sort -n | awk '{a[NR]=$1} END{n=NR; if(n){p50=a[int((n+1)/2)];
p90=a[int(0.9*n)]; printf("MAPQ p50=%.1f p90=%.1f n=%d\n",p50,p90,n)} else print "0 reads"}'
done
```

```
map.haplo.n64.gam -> MAPQ p50=60.0 p90=60.0 n=200000
map.haplo.n16.gam -> MAPQ p50=60.0 p90=60.0 n=200000
map.haplo.n128.gam -> MAPQ p50=60.0 p90=60.0 n=200000
```

Tutti gli allineamenti hanno confidenza massima, questo è perché le letture derivano da percorsi presenti nel grafo (problema previsto e controllato, perché verrà "risolto" nella prossima fase).

Tempi/RAM

```
grep -E "Elapsed \(wall clock\)|Maximum resident set size" logs/giraffe.haplo.*.log
```

```
logs/giraffe.haplo.n128.log:    Elapsed (wall clock) time (h:mm:ss or m:ss): 0:27.79
logs/giraffe.haplo.n128.log:    Maximum resident set size (kbytes): 3079928
logs/giraffe.haplo.n16.log:     Elapsed (wall clock) time (h:mm:ss or m:ss): 0:27.16
logs/giraffe.haplo.n16.log:     Maximum resident set size (kbytes): 3070640
logs/giraffe.haplo.n64.log:     Elapsed (wall clock) time (h:mm:ss or m:ss): 0:35.54
logs/giraffe.haplo.n64.log:     Maximum resident set size (kbytes): 3045996
```

Nessun impatto evidente del numero di path-cover sull'uso di memoria. Procediamo con l'esperimento con fastq reali.

Fase 5: “Allineamento di letture reali”

```
cd reads
```

Scaricamento dei file FASTQ reali: dati di sequenziamento FASTQ di altissima qualità dal Genome in a Bottle (GIAB) Consortium, specificamente del campione HG002 (Ashkenazim Trio son), uno dei campioni di riferimento più studiati al mondo. Ma soprattutto: HG002 NON è un individuo del 1000 Genomes Project.
Fonte:
https://ftp.ncbi.nlm.nih.gov/ReferenceSamples/giab/data/AshkenazimTrio/HG002_NA24385_son/NIST_HiSeq_HG002_Homogeneity-10953946/HG002_HiSeq300x_fastq/140528_D00360_0018_AH8VC6ADXX/Project_RM8391_RM8392/Sample_2A1/

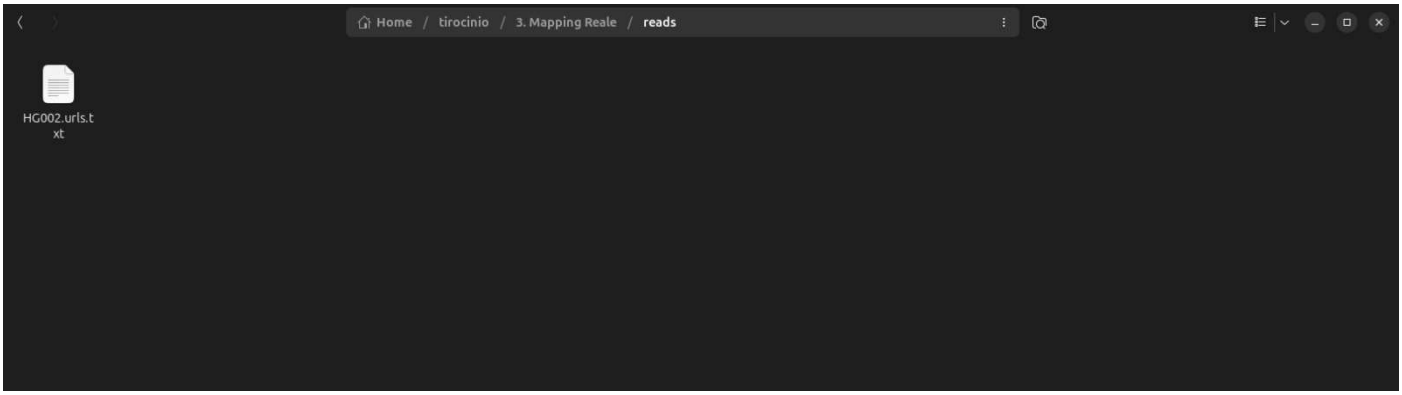
Index of /ReferenceSamples/giab/data/AshkenazimTrio/HG002_NA24385_son/NIST_HiSeq_HG002_Homogeneity-10953946/HG002_HiSeq300x_fastq/140528_D00360_0018_AH8VC6ADXX/Project_RM8391_RM8392/Sample_2A1

Name

Parent Directory
2A1_CGATGT_L001_R1_001.fastq.gz
2A1_CGATGT_L001_R1_002.fastq.gz
2A1_CGATGT_L001_R2_001.fastq.gz
2A1_CGATGT_L001_R2_002.fastq.gz
2A1_CGATGT_L002_R1_001.fastq.gz
2A1_CGATGT_L002_R1_002.fastq.gz
2A1_CGATGT_L002_R2_001.fastq.gz
2A1_CGATGT_L002_R2_002.fastq.gz
SampleSheet.csv

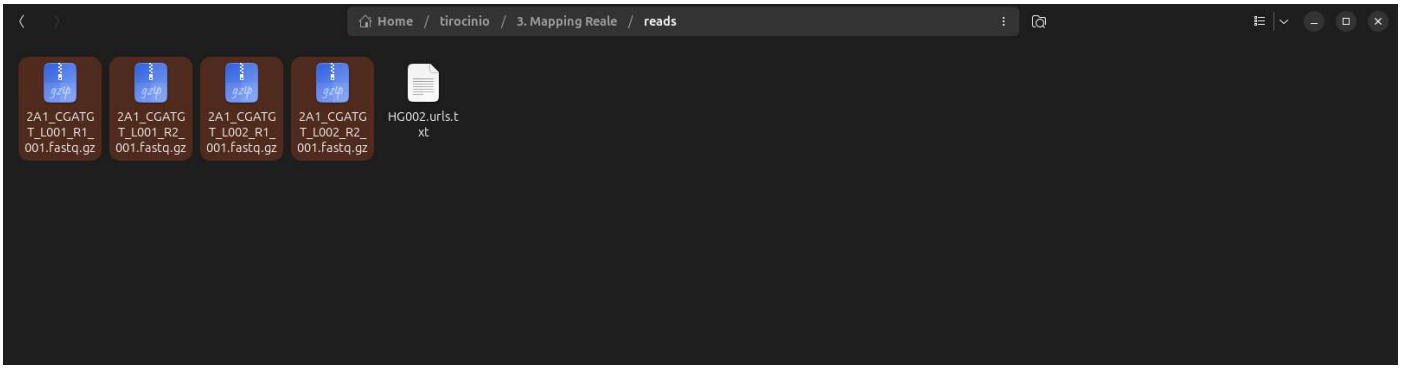
```
cat > HG002.urls.txt << 'EOF'
https://ftp.ncbi.nlm.nih.gov/ReferenceSamples/giab/data/AshkenazimTrio/HG002_NA24385_son/NIST_HiSeq_HG002_Homogeneity-10953946/HG002_HiSeq300x_fastq/140528_D00360_0018_AH8VC6ADXX/Project_RM8391_RM8392/Sample_2A1/2A1_CGATGT_L001_R1_001.fastq.gz

https://ftp.ncbi.nlm.nih.gov/ReferenceSamples/giab/data/AshkenazimTrio/HG002_NA24385_son/NIST_HiSeq_HG002_Homogeneity-10953946/HG002_HiSeq300x_fastq/140528_D00360_0018_AH8VC6ADXX/Project_RM8391_RM8392/Sample_2A1/2A1_CGATGT_L001_R2_001.fastq.gz

https://ftp.ncbi.nlm.nih.gov/ReferenceSamples/giab/data/AshkenazimTrio/HG002_NA24385_son/NIST_HiSeq_HG002_Homogeneity-10953946/HG002_HiSeq300x_fastq/140528_D00360_0018_AH8VC6ADXX/Project_RM8391_RM8392/Sample_2A1/2A1_CGATGT_L002_R1_001.fastq.gz

https://ftp.ncbi.nlm.nih.gov/ReferenceSamples/giab/data/AshkenazimTrio/HG002_NA24385_son/NIST_HiSeq_HG002_Homogeneity-10953946/HG002_HiSeq300x_fastq/140528_D00360_0018_AH8VC6ADXX/Project_RM8391_RM8392/Sample_2A1/2A1_CGATGT_L002_R2_001.fastq.gz

EOF
```

~ 976 bytes




```
xargs -n 1 -P 4 wget -c < HG002.urls.txt
```

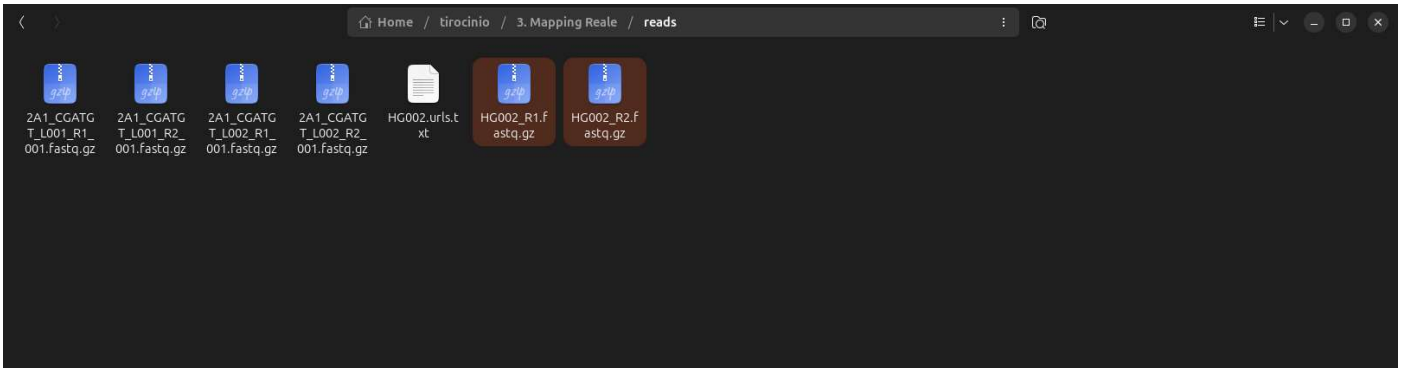
`xargs`: Prende l'input da un comando (in questo caso dal file HG002.urls.txt) e lo usa come argomento per un altro comando (wget)
`-n 1`: Dice a xargs di passare un solo URL per volta al comando wget
`-P 4`: L'opzione più importante per le prestazioni. Imposta il numero di processi paralleli a 4, permettendo di scaricare 4 file alla volta.
`wget -c`: Il comando per scaricare. L'opzione -c (continue) è molto utile, poiché permette di riprendere un download interrotto dal punto in cui si era fermato, invece di ricominciare da capo.
2A1_CGATGT_L001_R1_001.fastq.gz (~ 435.7 MB)
2A1_CGATGT_L001_R2_001.fastq.gz (~ 476.8 MB)
2A1_CGATGT_L002_R1_001.fastq.gz (~ 435.7 MB)
2A1_CGATGT_L002_R2_001.fastq.gz (~ 477.2 MB)



Unione dei file FASTQ per Lanes di Sequenziamento
Spesso, per gestire grandi volumi di dati, il sequenziatore divide i campioni in diverse "lanes" (ad esempio, L001, L002).
È necessario ricombinare tutte le lanes per avere un unico file R1 e un unico file R2 che rappresentino l'intero campione.

```
cat *L001_R1_001.fastq.gz *L002_R1_001.fastq.gz > HG002_R1.fastq.gz
cat *L001_R2_001.fastq.gz *L002_R2_001.fastq.gz > HG002_R2.fastq.gz
```

HG002_R1.fastq.gz (~ 871.4 MB)
HG002_R2.fastq.gz (~ 954 MB)



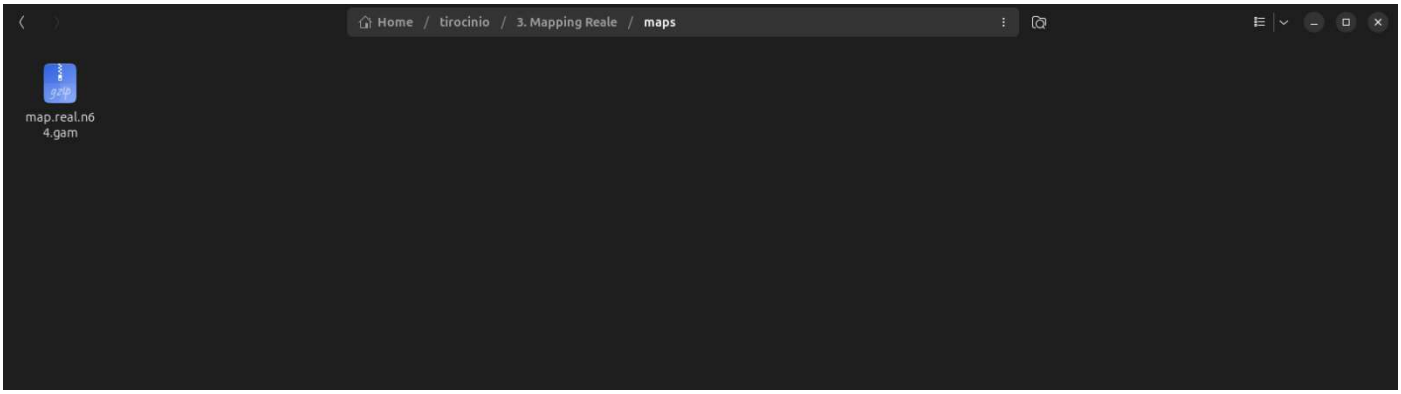

```
cd ..
```

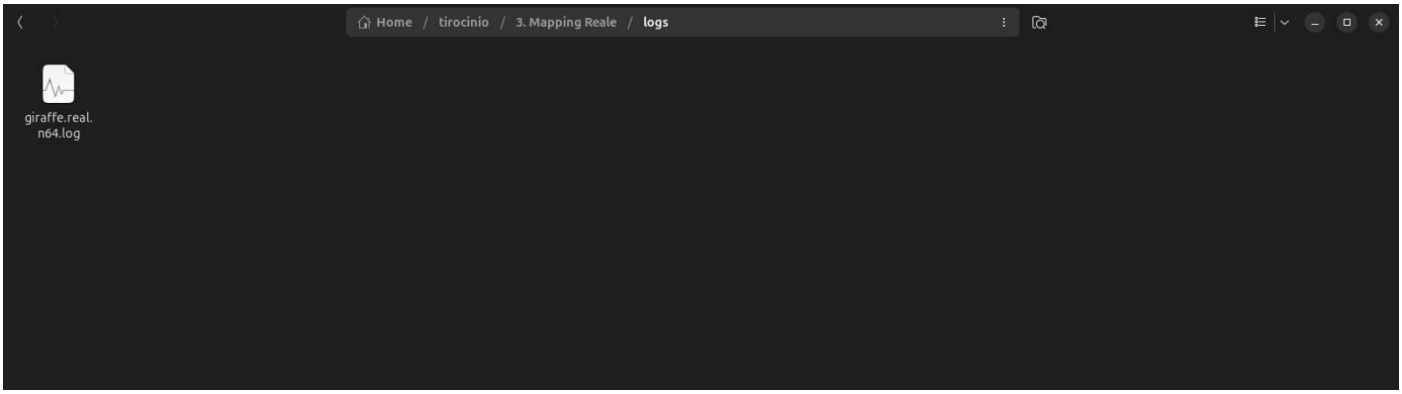
Mapping con Giraffe nelle tre configurazioni

standard (n=64)

```
(/usr/bin/time -v vg giraffe \
  -Z work/MHC.giraffe.gbz \
  -m work/MHC.shortread.withzip.min \
  -d work/MHC.dist \
  -f reads/HG002_R1.fastq.gz \
  -f reads/HG002_R2.fastq.gz \
  -t 16 -p > maps/map.real.n64.gam) 2> logs/giraffe.real.n64.log
```

map.real.n64.gam (~ 2.9 GB)
giraffe.real.n64.log (~ 4 kB)

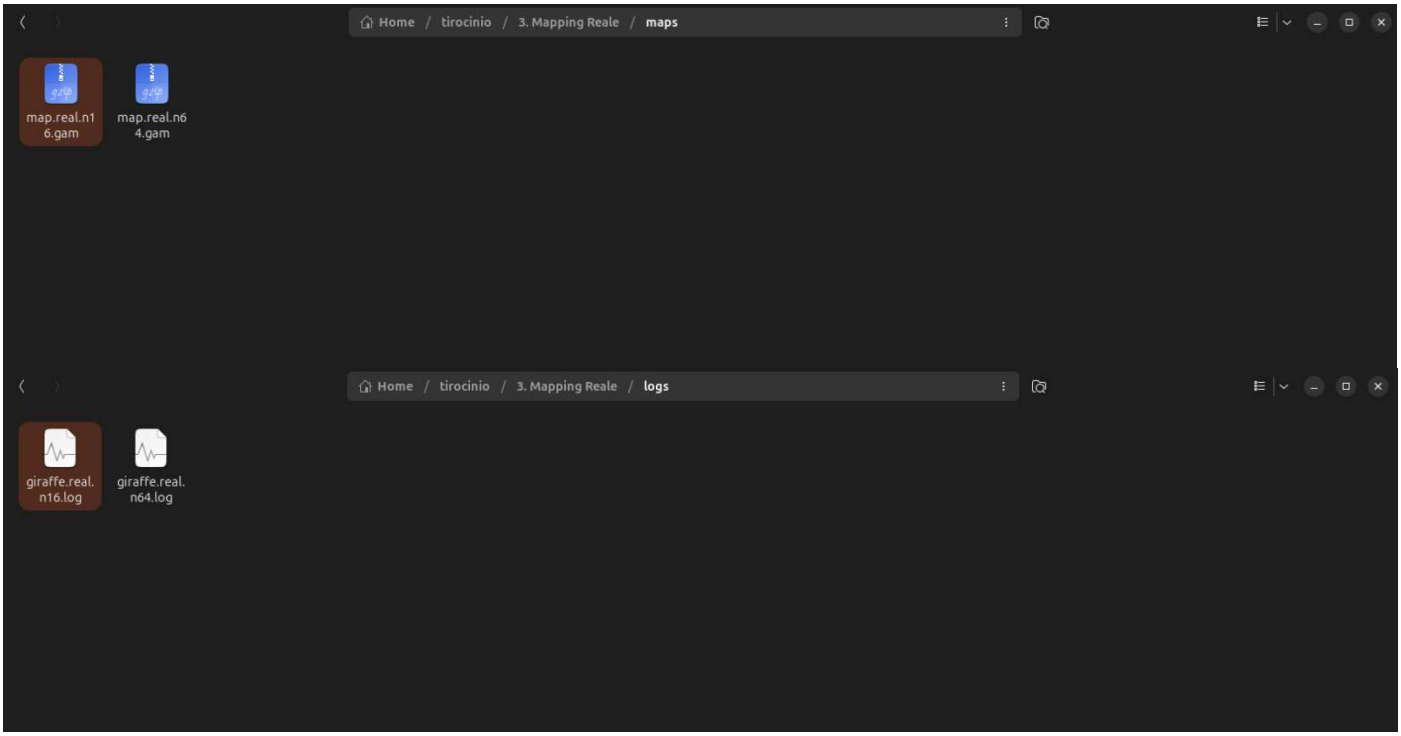




ridotta (n=16)

```
(/usr/bin/time -v vg giraffe \
  -Z work/MHC.n16.gbz \
  -m work/MHC.n16.shortread.withzip.min \
  -d work/MHC.dist \
  -f reads/HG002_R1.fastq.gz \
  -f reads/HG002_R2.fastq.gz \
  -t 16 -p > maps/map.real.n16.gam) 2> logs/giraffe.real.n16.log
```

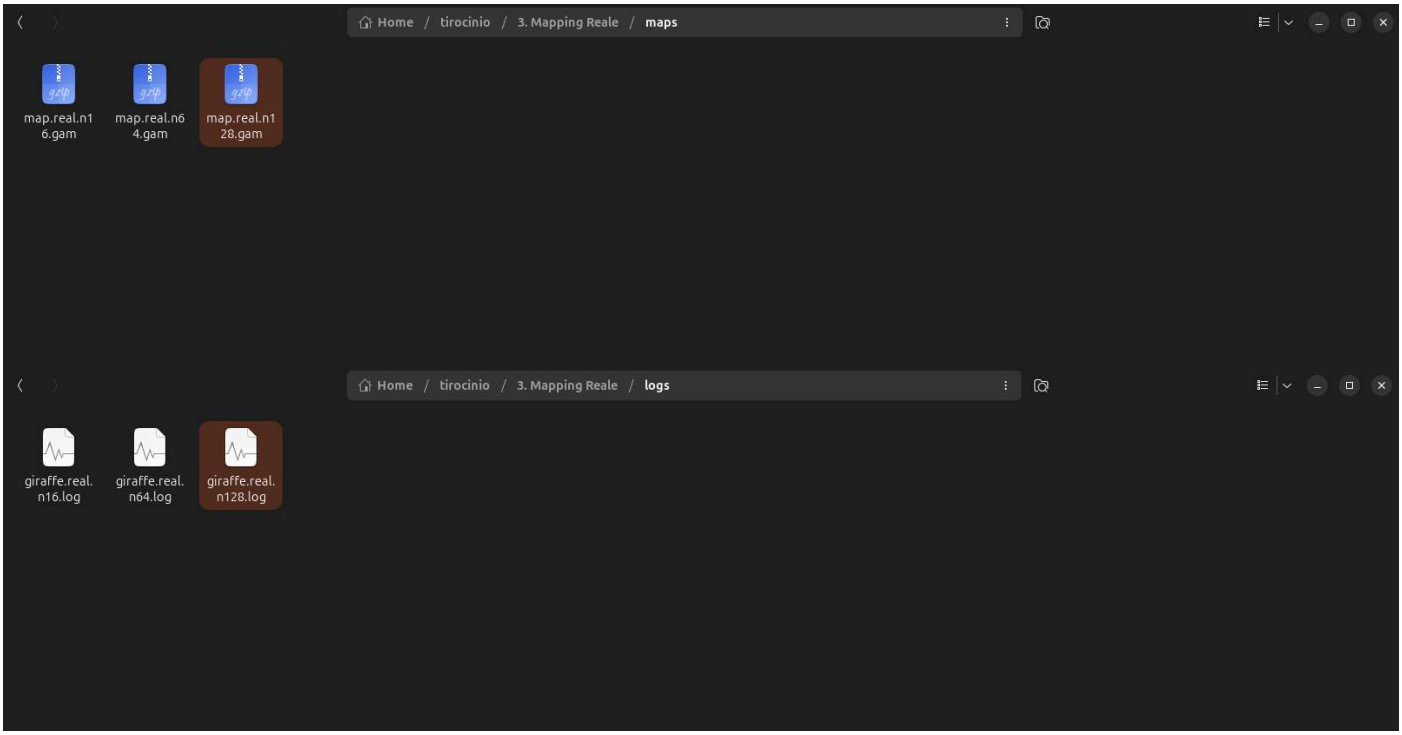
map.real.n16.gam (~ 2.9 GB)
giraffe.real.n16.log (~ 4 kB)



aumentata (n=128)

```
(/usr/bin/time -v vg giraffe \
  -Z work/MHC.n128.gbz \
  -m work/MHC.n128.shortread.withzip.min \
  -d work/MHC.dist \
  -f reads/HG002_R1.fastq.gz \
  -f reads/HG002_R2.fastq.gz \
  -t 16 -p > maps/map.real.n128.gam) 2> logs/giraffe.real.n128.log
```

map.real.n128.gam (~ 2.9 GB)
giraffe.real.n128.log (~ 4 kB)



Surject e indicizzazione dei file BAM

```
vg surject -p 6 -x work/MHC.xg -b maps/map.real.n64.gam  | samtools sort -o maps/map.real.n64.bam
samtools index maps/map.real.n64.bam
vg surject -p 6 -x work/MHC.xg -b maps/map.real.n16.gam  | samtools sort -o maps/map.real.n16.bam
samtools index maps/map.real.n16.bam
vg surject -p 6 -x work/MHC.xg -b maps/map.real.n128.gam  | samtools sort -o maps/map.real.n128.bam
samtools index maps/map.real.n128.bam
```

map.real.n16.bam (~ 1.8 GB), map.real.n16.bam.bai (~ 209.4 kB)
map.real.n64.bam (~ 1.8 GB), map. real.n64.bam.bai (~ 210.2 kB)
map. real.n128.bam (~ 1.8 GB), map. real.n128.bam.bai (~ 209.1 kB)

Definisco la regione sulla quale *samtools* dovrà lavorare

```
echo -e "6\t28477797\t33448354" > data/MHC.bed
```

~ 20 bytes

MQ0 fraction (ambiguità dell'allineamento): calcola la frazione di letture con qualità di mappatura pari a zero (MQ=0), cioè completamente ambigue nella regione MHC.

```
for B in maps/map.real.n64.bam maps/map.real.n16.bam maps/map.real.n128.bam; do
  echo -n "$(basename $B) -> "
  samtools view -F 2308 -L data/MHC.bed "$B" | \
    awk '{n++; if($5==0) z++} END{printf("MQ0_fraction=%.4f (n=%d)\n",(n?z/n:0),n)}'
done
```

```
map.real.n64.bam -> MQ0_fraction=0.6388 (n=193163)
map.real.n16.bam -> MQ0_fraction=0.6351 (n=189554)
map.real.n128.bam -> MQ0_fraction=0.6351 (n=189612)
```

~63% delle letture sono ambigue per tutti i valori di n → nessun miglioramento.

MAPQ distribution (qualità dell'allineamento): estrae le qualità di mappatura, calcola mediana (p50) e 90° percentile (p90).

```
for G in maps/map.real.n64.gam maps/map.real.n16.gam maps/map.real.n128.gam; do
```

```
  vg view -aj "$G" | jq -r '[.name, .mapping_quality, .score] | @tsv' > "stats/mapq.$(basename $G).tsv"
  echo -n "$(basename $G) -> "
  cut -f2 "stats/mapq.$(basename $G).tsv" | sort -n | awk '{a[NR]=$1} END{n=NR; if(n){p50=a[int((n+1)/2)];
p90=a[int(0.9*n)]; printf("MAPQ p50=%.1f p90=%.1f n=%d\n",p50,p90,n)} else print "0 reads"}'
done
```

```
map.real.n64.gam -> MAPQ p50=0.0 p90=9.0 n=16000000
map.real.n16.gam -> MAPQ p50=0.0 p90=9.0 n=16000000
map.real.n128.gam -> MAPQ p50=0.0 p90=9.0 n=16000000
```

la metà delle letture ha MAPQ=0 e quasi tutte rimangono comunque a bassa qualità. Nessun effetto del numero di path-cover.

Tempi/RAM

```
grep -E "Elapsed \(wall clock\)|Maximum resident set size" logs/giraffe.real.*.log
```

```
logs/giraffe.real.n128.log:	Elapsed (wall clock) time (h:mm:ss or m:ss): 19:41.91
logs/giraffe.real.n128.log:	Maximum resident set size (kbytes): 3515396
logs/giraffe.real.n16.log:	Elapsed (wall clock) time (h:mm:ss or m:ss): 20:12.03
logs/giraffe.real.n16.log:	Maximum resident set size (kbytes): 3511512
logs/giraffe.real.n64.log:	Elapsed (wall clock) time (h:mm:ss or m:ss): 19:27.81
logs/giraffe.real.n64.log:	Maximum resident set size (kbytes): 3522840
```

tempi e RAM praticamente identici → il numero di path-cover non incide sulle risorse.

Flagstat: riassunto veloce delle letture

```
for B in maps/map.real.n64.bam maps/map.real.n16.bam maps/map.real.n128.bam; do
  echo "== $(basename $B) =="; samtools flagstat "$B" | sed -n '1,8p'
done
```

```
== map.real.n64.bam ==
16000000 + 0 in total (QC-passed reads + QC-failed reads)
16000000 + 0 primary
0 + 0 secondary
0 + 0 supplementary
0 + 0 duplicates
0 + 0 primary duplicates
4477803 + 0 mapped (27.99% : N/A)
4477803 + 0 primary mapped (27.99% : N/A)

== map.real.n16.bam ==
16000000 + 0 in total (QC-passed reads + QC-failed reads)
16000000 + 0 primary
0 + 0 secondary
0 + 0 supplementary
0 + 0 duplicates
0 + 0 primary duplicates
4478020 + 0 mapped (27.99% : N/A)
4478020 + 0 primary mapped (27.99% : N/A)

== map.real.n128.bam ==
16000000 + 0 in total (QC-passed reads + QC-failed reads)
16000000 + 0 primary
0 + 0 secondary
0 + 0 supplementary
0 + 0 duplicates
0 + 0 primary duplicates
4478026 + 0 mapped (27.99% : N/A)
4478026 + 0 primary mapped (27.99% : N/A)
```

percentuale di letture mappate identica per n=16, 64, 128 → nessun miglioramento.

Conclusione dell'esperimento

Mi aspettavo che aumentando il numero di percorsi nella GBWT (path-cover n=16, 64, 128) la qualità dell'allineamento potesse migliorare, perché in teoria una GBWT più ricca dovrebbe fornire a Giraffe più alternative lungo cui instradare correttamente le letture. Tuttavia, nei risultati ottenuti, non è emerso alcun miglioramento significativo.

Perché non è successo?

La spiegazione risiede nel modo in cui viene costruita la GBWT all'interno del workflow ufficiale di VG.
Il comando:

```
vg autoindex --workflow giraffe \
  -r data/GRCh38.chr6_as6.fa \
  -v data/MHC.1000G.phased.vcf.gz \
  -p work/MHC \
```

costruisce correttamente il grafo pangenomico (includendo varianti reali del 1000 Genomes), ma non ricostruisce gli aplotipi reali degli individui presenti nel VCF.
Al loro posto, genera automaticamente dei path-cover, cioè percorsi sintetici che servono a "coprire" il grafo in modo completo e regolare dal punto di vista topologico. Questi percorsi non rappresentano individui reali e non imitano la struttura genetica della popolazione.
Di conseguenza, aumentando n da 16 a 64 a 128 non si sta aumentando l'informazione biologica, ma solo la quantità di percorsi artificiali. È quindi normale che l'allineamento non migliori: non stiamo aggiungendo haplotipi reali, ma semplicemente versioni sempre più dense dello stesso costrutto sintetico.

Perché non vengono utilizzati gli aplotipi reali?

Sebbene il VCF sia *phased* e contenga informazioni sugli alleli, esso non fornisce sequenze aplotipiche continue. Specialmente nella regione MHC, mancano interi tratti di fase, ci sono varianti filtrate, buchi, ambiguità e dati incompleti. Ricostruire aplotipi reali da questi file richiederebbe pipeline molto più complesse, con grande potenza di calcolo e algoritmi avanzati: è il tipo di lavoro svolto da grandi iniziative internazionali come l'Human Pangenome Project.

Comparing methods for constructing and representing human pangenome graphs

contains a list of variations from it.
Many human pangenomes have been generated, e.g. using Pantools [10] (7 genomes), `Minigraph` [14] (94 haplotypes), `Minigraph-Cactus` [16, 17] and `pggb` [8] (94 single chromosomes), and `TwoPaCo` [13] (100 simulated genomes). Lastly, a draft version of a human reference pangenome constructed using `pggb` and the `Minigraph-Cactus` pipeline has appeared in a very recent article from the Human Pangenome Reference Consortium [8]. These pangenomes are still limited by some factors: at the present moment, the number of high-quality haplotype assemblies is still low, even if it is expected to grow in the future; the vcf files containing variation are limited in terms of bias, type of variation or number of samples; the population representation, even if opened up in recent years to more ethnicities, is still affected by sampling bias.

Per questo motivo, VG adotta una scelta conservativa: invece di tentare una ricostruzione potenzialmente errata o troppo pesante, costruisce percorsi sintetici robusti e garantiti dal punto di vista computazionale.

Cosa sarebbe servito per osservare un reale miglioramento con la GBWT?

Per sfruttare davvero la GBWT nella regione MHC sarebbero necessari:

- aplotipi reali completi (non disponibili nella versione del 1000 Genomes utilizzata),
- oppure un grafo personalizzato costruito appositamente per l'individuo studiato,
- oppure l'uso di pipeline avanzate basate su cluster HPC per ricostruire sequenze aplotipiche continue.

Nessuna di queste possibilità è realisticamente praticabile disponendo di risorse limitate.

A draft human pangenome reference

Here the Human Pangenome Reference Consortium presents a first draft of the human pangenome reference. The pangenome contains 47 phased, diploid assemblies from a cohort of genetically diverse individuals[1]. These assemblies cover more than 99% of the expected sequence in each genome and are more than 99% accurate at the structural and base pair levels. Based on alignments of the assemblies, we generate a draft pangenome that captures known variants and haplotypes and reveals new alleles at structurally complex loci. We also add 119 million base pairs of euchromatic polymorphic sequences and 1,115 gene duplications relative to the existing reference GRCh38. Roughly 90 million of the additional base pairs are derived from structural variation. Using our draft pangenome to analyse short-read data reduced small variant discovery errors by 34% and increased the number of structural variants detected per haplotype by 104% compared with GRCh38-based workflows, which enabled the typing of the vast majority of structural variant alleles per sample.

Quindi a cosa è servito questo lavoro?

L'esperimento dimostra che un pangenoma costruito solo con varianti popolazionali e path-cover sintetici non è sufficiente per migliorare l'allineamento di un individuo reale, soprattutto in regioni estremamente variabili come la MHC.

La pipeline autoindex è ottima per efficienza, riproducibilità e stabilità, ma non può sostituire l'informazione biologica contenuta in veri haplotipi continui.

In altre parole, l'esperimento mostra chiaramente dove il pangenoma funziona, dove no, e quali condizioni sono necessarie perché la GBWT possa davvero fare la differenza.

Fase 6: “Allineamento lineare”

L’obiettivo di questa ultima fase è allineare le letture reali di HG002 sulla sequenza lineare del cromosoma 6 (GRCh38), estratta nella stessa regione MHC del resto degli esperimenti.

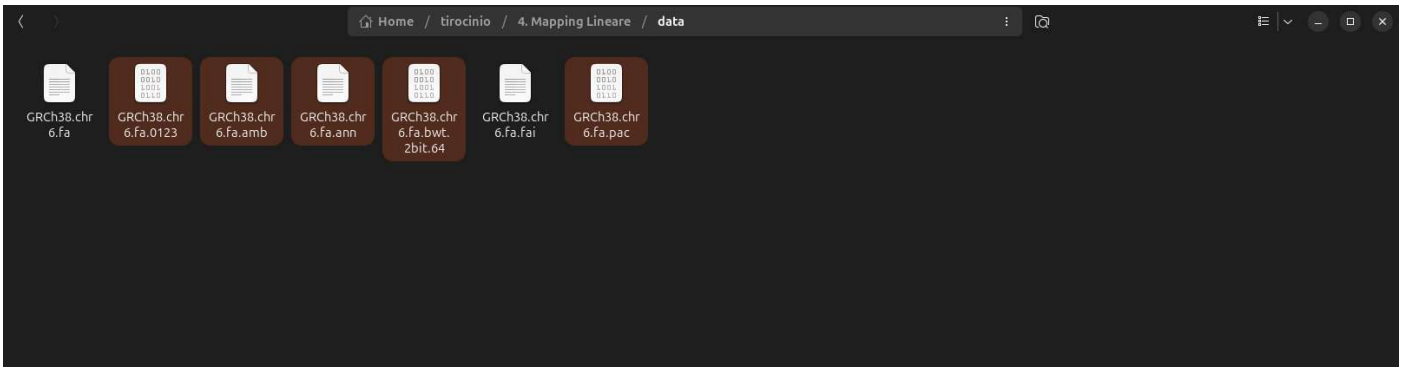
```
conda install -c bioconda bwa-mem2
```

```
.
├── data
│   ├── GRCh38.chr6.fa
│   └── GRCh38.chr6.fa.fai
├── logs
├── maps
├── reads
│   ├── HG002_R1.fastq.gz
│   └── HG002_R2.fastq.gz
├── stats
└── work
```

Indicizzazione del riferimento: costruisce le strutture dati necessarie per allineare rapidamente le letture al riferimento lineare.

```
bwa-mem2 index data/GRCh38.chr6.fa
```

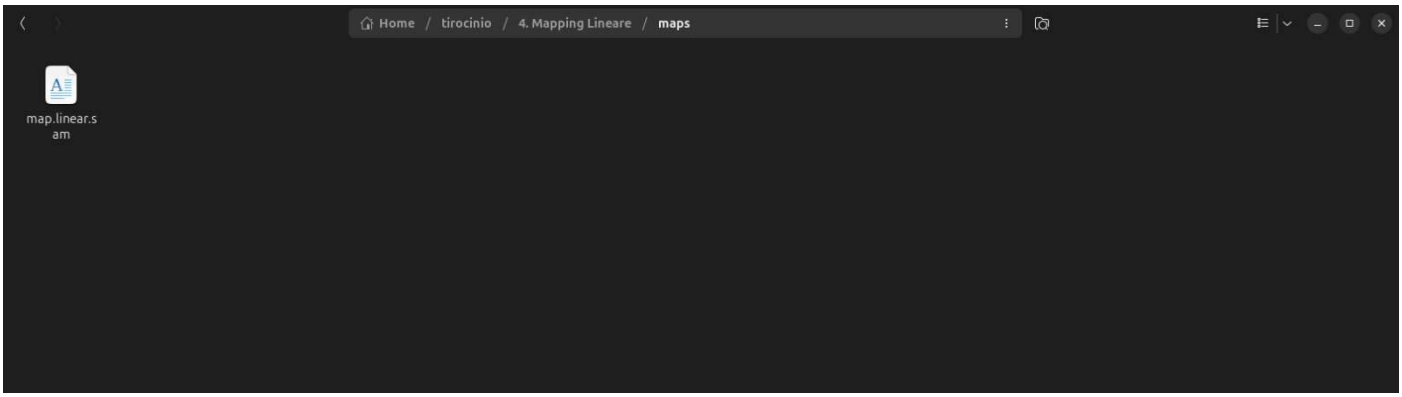
GRCh38.chr6.fa.0123 (~ 341.6 MB)
GRCh38.chr6.fa.amb (~ 262 bytes)
GRCh38.chr6.fa.ann (~ 44 bytes)
GRCh38.chr6.fa.bwt.2bit.64 (~ 555.1 MB)
GRCh38.chr6.fa.pac (~ 42.7 MB)

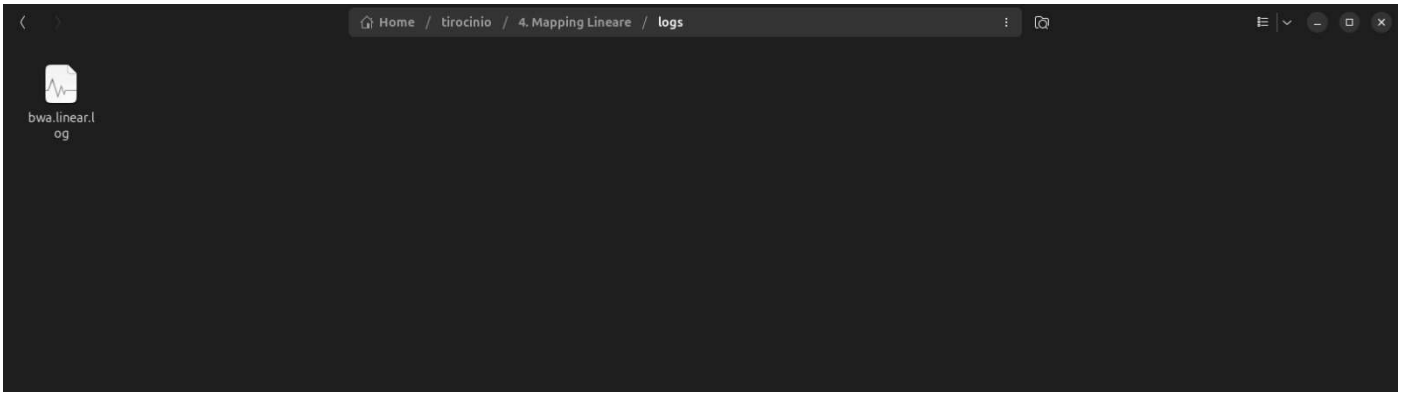


Allineamento lineare

```
/usr/bin/time -v bwa-mem2 mem -t 16 \
  data/GRCh38.chr6.fa \
  reads/HG002_R1.fastq.gz \
  reads/HG002_R2.fastq.gz \
  > maps/map.linear.sam 2> logs/bwa.linear.log
```

map.linear.sam (~ 6.7 GB)
bwa.linear.log (~ 21 kB)

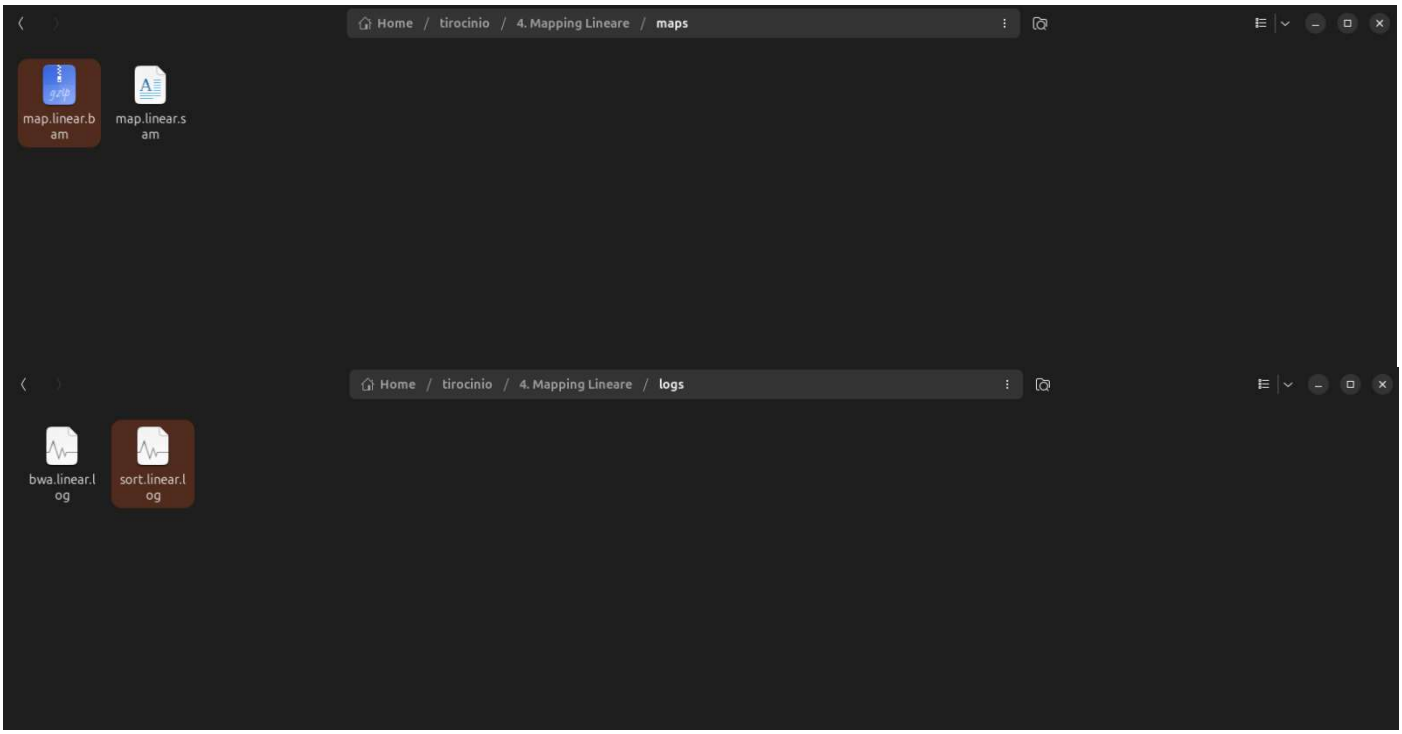




Conversione SAM > BAM

```
/usr/bin/time -v samtools sort \
  -@ 4 \
  -m 512M \
  -o maps/map.linear.bam \
  maps/map.linear.sam \
  2> logs/sort.linear.log
```

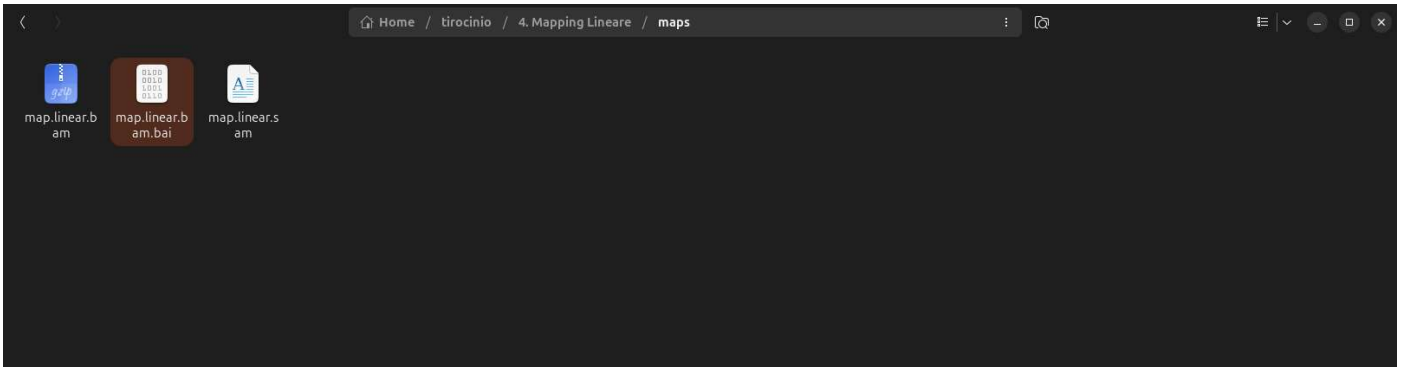
map.linear.bam (~ 2.2 GB)
sort.linear.log (~ 877 bytes)



Indicizzazione

```
samtools index maps/map.linear.bam
```

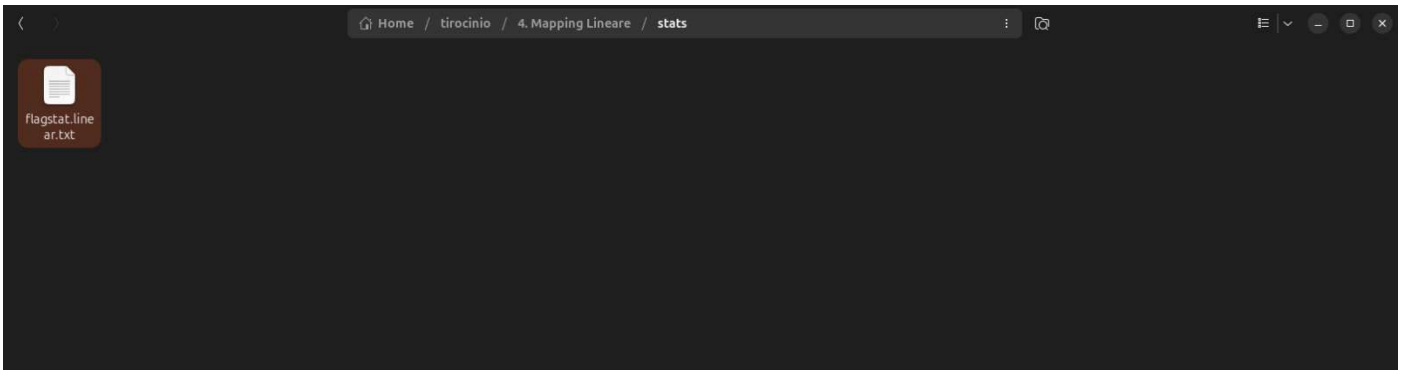
map.linear.bam.bai (~ 361.7 kB)



Flagstat: per ottenere un quadro generale del numero di letture mappate, duplicati, supplementary ecc.

```
samtools flagstat maps/map.linear.bam > stats/flagstat.linear.txt
```

flagstat.linear.txt (~ 522 bytes)



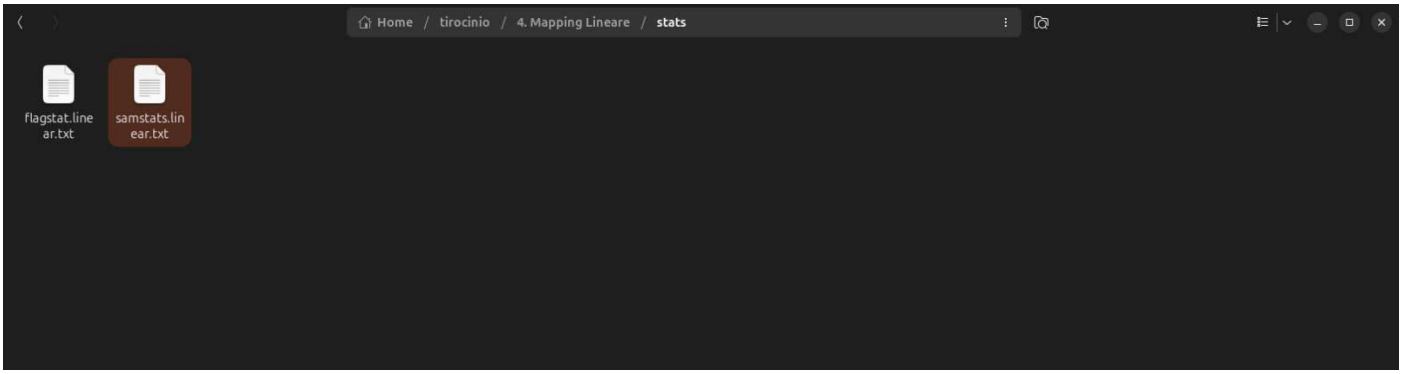

```
sed -n '1,8p' stats/flagstat.linear.txt
```
```
16754652 + 0 in total (QC-passed reads + QC-failed reads)
16000000 + 0 primary
0 + 0 secondary
754652 + 0 supplementary
0 + 0 duplicates
0 + 0 primary duplicates
6742392 + 0 mapped (40.24% : N/A)
5987740 + 0 primary mapped (37.42% : N/A)
```

Circa il 37% delle letture si allinea su chr6, cifra coerente con la percentuale reale di letture che appartengono a questo cromosoma.

Stats

```
samtools stats maps/map.linear.bam > stats/samstats.linear.txt
```

samstats.linear.txt (~ 262.9 kB)



Definisco la regione sulla quale *samtools* dovrà lavorare

```
echo -e "chr6\t28477797\t33448354" > data/MHC.bed
```

MQ0 fraction (ambiguità dell'allineamento)

```
samtools view -F 2308 -L data/MHC.bed maps/map.linear.bam \
  | awk '{n++; if($5==0) z++} END{
      printf("MQ0_fraction=%.4f (n=%d)\n",(n?z/n:0),n)
  }'
```
```
MQ0_fraction=0.3908 (n=187379)
```

Circa il 39% degli allineamenti nella MHC è ambiguo (MAPQ=0).
È molto meno del grafo Giraffe (≈63%), segno che in questa regione altamente complessa il grafo usato non riesce a risolvere la variazione strutturale.

Distribuzione MAPQ nella MHC: capire la qualità dell'allineamento tra le letture non ambigue.

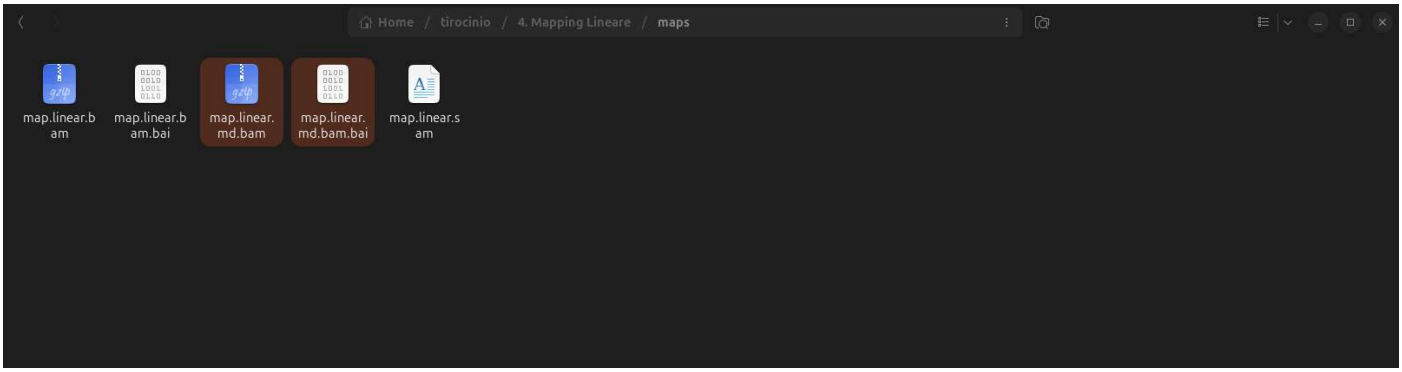
```
samtools view -F 2308 -L data/MHC.bed maps/map.linear.bam \
  | awk '{print $5}' | sort -n \
  | awk '{
      a[NR]=$1
    }
    END{
      n=NR
      if(n==0){print "MAPQ: nessuna read"; exit}
      p50=a[int((n+1)/2)]
```

```
    p90=a[int(0.9*n)]
    printf("LINEARE → MAPQ p50=%.1f p90=%.1f n=%d\n", p50, p90, n)
  }'
LINEARE → MAPQ p50=5.0 p90=60.0 n=187379
```

p50=5 → metà delle letture hanno confidenza moderata
p90=60 → il top 10% è ad altissima confidenza
Anche qui: BWA-mem2 batte nettamente Giraffe (p50=0, p90=9).

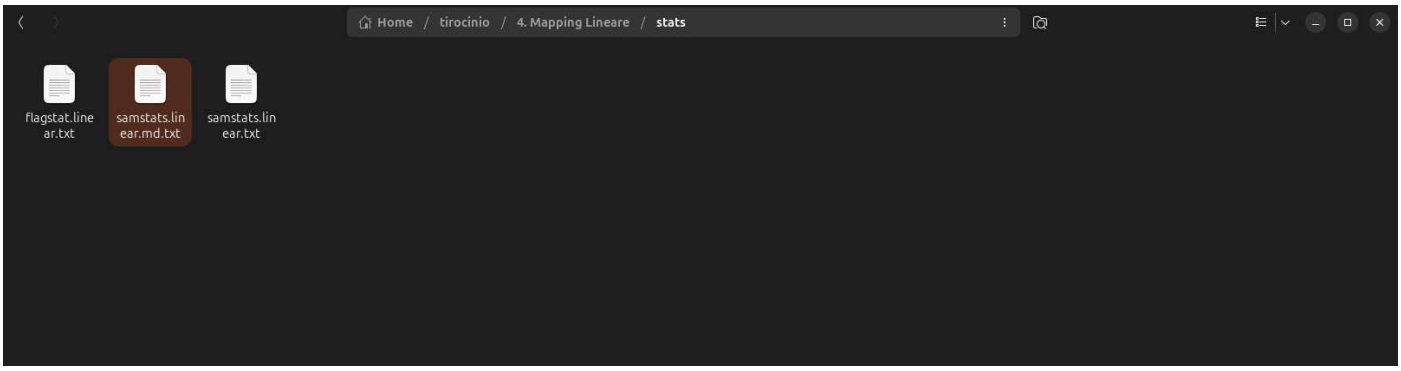
```
samtools calmd -b maps/map.linear.bam data/GRCh38.chr6.fa > maps/map.linear.md.bam
samtools index maps/map.linear.md.bam
```

map.linear.md.bam (~ 2.2 GB)
map.linear.md.bam.bai (~ 361.9 kB)




```
samtools stats maps/map.linear.md.bam > stats/samstats.linear.md.txt
```

samstats.linear.md.txt (~ 262.9 kB)



Error rate

```
awk -F'\t' '$1=="SN" && $2=="error rate:"{print}' stats/samstats.linear.md.txt
SN	error rate:	5.115142e-02	# mismatches / bases mapped (cigar)
```

Un mismatch rate del ~5% è realistico per:

- differenze reali tra HG002 e GRCh38
- errori di sequenziamento
- complessità della MHC

Valore assolutamente coerente per la regione.

Tempi e consumo RAM

```
grep -E "Elapsed \(wall clock\)|Maximum resident set size" logs/bwa.linear.log logs/sort.linear.log
logs/bwa.linear.log:	Elapsed (wall clock) time (h:mm:ss or m:ss): 44:03.76
logs/bwa.linear.log:	Maximum resident set size (kbytes): 6787996
logs/sort.linear.log:	Elapsed (wall clock) time (h:mm:ss or m:ss): 2:24.06
logs/sort.linear.log:	Maximum resident set size (kbytes): 2378012
```

BWA-mem2: ~44 min, 6.8 GB RAM
samtools sort: ~2.4 GB RAM
Totale ~46 minuti

Giraffe era più veloce (~19–20 min) e più leggero (~3.5 GB),
ma la qualità dell'allineamento nella MHC era peggiore.

Chiusura

Nella regione MHC, una tra le più complesse e variabili del genoma umano, il mapping lineare mostra un comportamento più stabile rispetto a Giraffe quando la GBWT è costruita soltanto tramite path-cover sintetici. In particolare, il lineare produce una minore frazione di allineamenti completamente ambigui (MQ0), una distribuzione MAPQ più convincente e un mismatch rate coerente con le differenze biologiche tra HG002 e il riferimento GRCh38.

Di contro, Giraffe è più rapido ed efficiente in termini di memoria, ma l'assenza di aplotipi reali nella GBWT compromette il suo principale punto di forza in regioni altamente variabili: la capacità di seguire fedelmente i percorsi plausibili nel grafo. Questo confronto dimostra quindi che la qualità del mapping pangenomico dipende in modo critico dalla qualità della GBWT: se non contiene aplotipi reali, l'aumento del numero di path-cover non si traduce in un miglioramento dell'allineamento, mentre un riferimento lineare consolidato continua a mantenere prestazioni più affidabili. In sintesi, il pangenoma offre potenziale enorme, ma la sua efficacia dipende soprattutto dalla disponibilità di aplotipi reali e accurati; in loro assenza, il mapping lineare resta sorprendentemente competitivo.