Treball de Fi de Grau

GENÒMICA COMPUTACIONAL

Autor:

Sergi Alcaide i Portet

Ponent:

David Carrera Pérez

Codirectors:

David Torrents Arenales

Friman Sánchez Castaño

12 d'octubre de 2015

Contingut

1 Introducció 4
1.1 Part biològica 4
1.2 Motivació del projecte 6
1.3 Objectius 7
1.4 Estat de l'art 8
1.4.1 Antecedents en mutacions somàtiques 8
1.4.2 Estat actual d'estudis poblacionals 8
1.4.3 Eines utilitzades: *MPI, Suffix-Tree, C++* 9
1.5 Competències 10
2 Planificació 12
2.1 Descripció de les tasques 12
2.2 Explicitació de l'ordre lògic de les tasques 16
2.3 Diagrama de Gantt 17
2.4 Anàlisi de les modificacions en la planificació inicial 19
2.5 Valoració d'alternatives i pla d'acció 19
3 Pressupost 21
3.1 Identificació i estimació dels costos 21
4 Cos del treball 26
4.1 Flux de dades de l'aplicació 26
4.1.1 *Suffix Tree* 26
4.1.2 Flux de dades 28
4.2 Creant la 1a versió 28
4.2.1 Algoritme general de l'aplicació 28
4.2.2 Estructura de l'aplicació 29
4.3 Anàlisi 40
4.4 Millores 48
4.4.1 *Master* amb *Suffix Tree (Master Tree)* 48
4.4.2 Partitions 49
4.4.3 Reverse 52
4.4.4 Lectura d'arxius comprimits 54
4.4.5 Comunicació *Master-Slaves* 54
4.5 Resultats 55
4.6 Estudis poblacionals 57
5 Conclusions 59
5.1 Conclusions generals 59

5.2 Ètica del projecte i regulacions adherides 60
5.3 Sostenibilitat i compromís social 60
5.3.1 Dimensió econòmica 60
5.3.2 Dimensió social 60
5.3.3 Dimensió ambiental 61
6 Referències 62

1 Introducció

1.1 Part biològica

Avui en dia una de les preocupacions en quant a salut que més incideixen en la població és la del càncer, això ha provocat l'augment de recursos destinats a la investigació en aquest camp. Quins canvis es produeixen en el genoma d'un individu que provoquin càncer, com es poden evitar aquests canvis... Moltes preguntes han sorgit en aquest camp i són molts els científics que intenten donar-hi resposta.

Una de les estratègies que s'està emprant més en els estudis actuals de la genòmica del càncer es basa en el principi de comparar genomes de cèl·lules sanes amb cèl·lules tumorals del mateix pacient per tal d'identificar els canvis (mutacions somàtiques) que s'han produït en el genoma tumoral que puguin ser responsables de la malaltia. Per això, es prenen dues mostres del pacient, una d'una zona sana i l'altra d'una zona afectada pel càncer, i se'n extreu l'ADN (àcid desoxiribonucleic) per procedir a la seva lectura, és a dir, la seva seqüenciació[1].

El genoma humà està compost per quatre nucleòtids: adenina, guanina, citosina i timina, simbolitzats per les lletres A, G, C, i T, respectivament. El procés de seqüenciació converteix les molècules d'ADN en milers de milions de lectures (text) que es recullen en fitxers de format fast queue (.fq) on, a més de les subseqüències extretes també tenim informació que ens indica la qualitat/fiabilitat de cada mostra. Per tal de neutralitzar els errors de seqüenciació que puguem induir del procés (que no és perfecte), el procés de lectura cobreix tot el genoma moltes vegades (unes 30 o 60, també anomenat *coverage*[2]), amb la conseqüència directa que el nombre de dades també es multiplica amb la mateixa constant.

Tenint en compte que el genoma humà consta de 3,2·109 lletres (nucleòtids) aproximadament i que s'esperen no més de 10.000 mutacions en cada tumor, la identificació d'aquests canvis suposa un repte computacional important a nivell algorísmic i d'implementació. A més cal considerar que les subseqüències que llegim del genoma no estan ordenades i per tant tenim milers de milions de subseqüències desordenades i que estan replicades vàries vegades (degut al *coverage*). A més a més, si considerem que les millores en les tècniques de seqüenciació estan facilitant aquest procés i generant milers de genomes seqüenciats, els protocols d'anàlisis de seqüència també han de considerar, més enllà de la detecció fiable de mutacions, que aquest procés sigui escalable i que arribi a un nivell de paral·lelització que permeti processar centenars de mostres a la vegada. D'aquesta manera un problema que era inicialment de caràcter biològic passa a ser un problema de caràcter computacional.

Per trobar les mutacions somàtiques es requereixen dos genomes per tant 6.400 milions de bases que acostumen a tenir *coverage* d'entre 30 i 60 (384 GBytes[3]). Per poder fer l'anàlisi a més de la informació essencial (subseqüències) es requereix també informació addicional que

---

[1]Les tècniques de seqüenciació llegeixen la cadena d'ADN de la mostra i mitjançant la reacció que ofereix cada punt de la mostra a un estímul lumínic es classifica en els diversos tipus de nucleòtids (adenina, citocina, guanina o timina). A més a més assigna a cada nucleòtid un caràcter ascii que indica la fiabilitat de la classificació.

[2] Mot que s'empra per indicar el nombre de vegades que ha estat seqüenciat cada genoma. El procés de seqüenciació no és un procés exacte i produeixen errors durant l'extracció de les bases. Per aquesta raó normalment els genomes se seqüencien múltiples vegades per poder eliminar/filtrar aquests errors.

[3] Comptant que cada caràcter s'emmagatzema en 8 bits (1 Byte) i cada genoma té 3.200 milions de bases i un *coverage* de 60.

relaciona, comptabilitza i guarda informació associada (com són un nom que identifica la subseqüència, nosaltres no la utilitzarem, i la qualitat de la mostra) . Com es pot comprovar el problema que es pretén tractar és un problema complex especialment pel que fa el volum de dades.

Malgrat s'han invertit molts esforços en trobar solucions a aquest problema i s'han generat nombrosos programes per la detecció de mutacions somàtiques, aquest procés encara no esta resolt i segueix suposant un repte important per la biomedicina. De fet, amb la facilitat de seqüenciació, l'anàlisi d'aquestes seqüencies s'ha convertit en el coll d'ampolla dels estudis de les malalties a nivell genòmic. Pràcticament tots els programes existents es basen en la comparació de les seqüencies del pacient contra un genoma de referència. Aquest pas, malgrat facilita el procés, també comporta unes limitacions inherents al procés d'alineament, fent que no s'assoleixin nivells de fiabilitat suficients per la majoria d'estudis, o per l'aplicació d'aquesta metodologia a la clínica[4]. Altres alternatives, com el programa SMUFIN *(Somatic Mutation Finder)*, es basen en un altre tipus de processament primari de les dades, el que resulta en una millor detecció de les mutacions. Degut a que aquests programes s'han realitzat en un entorn estrictament biològic, les implementacions existents encara deixen espai per moltes millores que requereixen solucions més computacionals, també en quant a trobar la millor combinació software-hardware.

Aquest projecte es basa en el disseny i implementació de solucions alternatives per la identificació de mutacions somàtiques a partir de la comparació de dos genomes, un sa i un altre malalt. Per això hem escollit, com a base, parts de l'estratègia utilitzada pel programa SMUFIN, desenvolupat al grup de genòmica computacional del BSC (*Barcelona Supercomputing Center*). Malgrat aquest programa ja ha superat a altres existents en rapidesa i en fiabilitat, encara queden molts punts per millorar-lo. A més, gran part del codi d'SMUFIN no és interpretable ni avaluable, probablement degut a les pressions de temps entre les que es va generar, i té moltes dependències externes pel que fa a llibreries.

Un altre camp que s'està potenciant molt últimament és el d'estudis poblacionals, on es busquen quines similituds i diferències tenim dins d'una mateixa població. Una gran mostra de la importància que està tenint aquest camp és el gran nombre de projectes de seqüenciació que existeixen actualment com UK10K [1], GoNL [2] o el famós projecte dels Estats Units "*Precision Medicine*" amb l'objectiu de seqüenciar un milió de voluntaris (amb un pressupost inicial de 215 milions de dòlars).

Nosaltres també som conscients de la importància d'aquest camp i creiem que en el nostre cas també podem aplicar un algoritme semblant al de *SMUFIN* (en quan a l'ús del *Suffix Tree)*, que pugui analitzar aquestes dades. És per això que també ens plantegem l'objectiu de que la nostra aplicació també serveixi per a fer estudis poblacionals, el que representa un augment del nombre de genomes que haurem d'introduir com a entrada i per tant un augment molt important pel que fa el nombre de dades que utilitzarà el programa. A més el canvi que es fa respecte les mutacions somàtiques és que haurem de contraposar tots els genomes contra els altres i que hi haurà moltes més divergències que en el cas d'un mateix individu, degut a la dimensió d'aquest altre objectiu i la limitació de temps ens centrarem en aconseguir fer una versió senzilla que validi l'ús del *Suffix Tree* en estudis poblacionals.

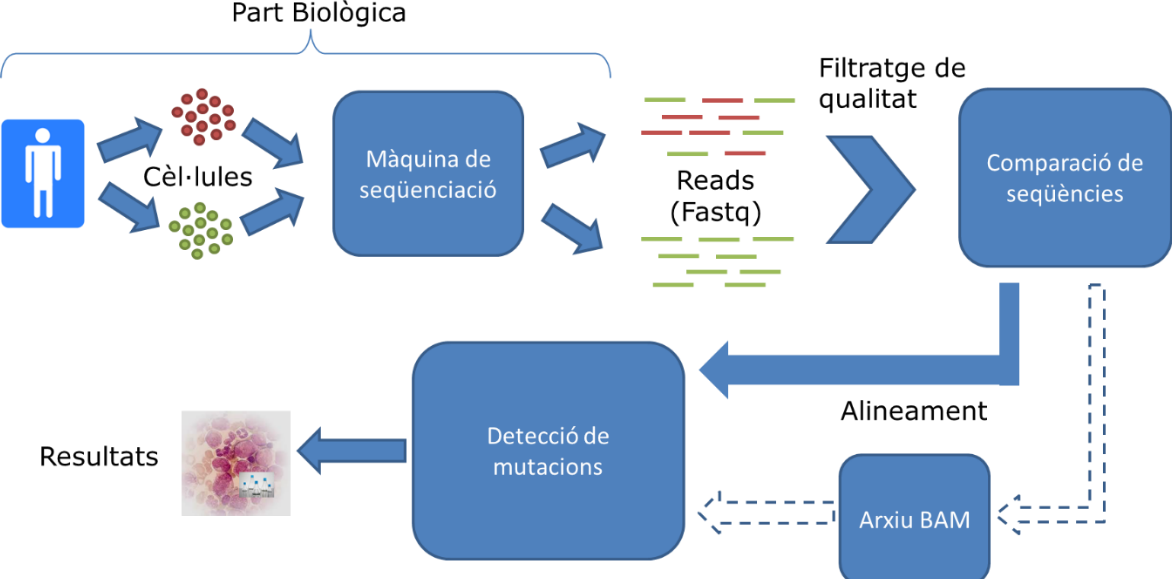
[4] Sobretot degut a que al contraposar-lo amb el genoma de referència, tenim moltes altres variacions i per tant es complica encara més el trobar les diferències entre sa i tumoral.

1.2 Motivació del projecte

L'*SMUFIN* és una aplicació desenvolupada al *BSC* (Barcelona Supercomputing Center), concretament al grup de *Computational Genomics*, que es basa en el principi de comparar els genomes sans i tumorals d'un mateix pacient per tal de trobar les mutacions somàtiques que puguin provocar el càncer i no utilitzar el genoma de referència per alinear-los. Per a cada mutació trobada n'indica de quin tipus és i la posició dins el genoma sencer.

En el moment de la seva publicació [3], va demostrar ser un dels pioners en aquesta estratègia amb les bones estadístiques que s'indiquen en l'article, tant en especificitat, sensitivitat i rapidesa, especialment en les mutacions de medul·loblastomes en nens, leucèmia mieloide aguda, càncer de pròstata i limfomes.

El flux de dades general de *SMUFIN* és el següent:



Il·lustració 1: Flux de dades simplificat del programa SMUFIN

La part estrictament biològica com hem comentat consisteix en agafar cèl·lules sanes i tumorals d'un pacient i seqüenciar-les, amb això obtenim fitxers en format *fastq* dels dos tipus de cèl·lules:

El format *fastq* és el següent:

Per a cada *read* ens trobem quatre línies consecutives:

1. La primera és un nom que indica el *read* per exemple: HW6217
2. La segona és el *read* pròpiament dit, i que conté els nucleòtids (identificats per les quatre lletres A, C ,G i T): ACGTAGTCAGCTA

3. La tercera és un signe de suma: +

4. Per acabar tenim una línia amb el mateix nombre de caràcters que la segona que ens indiquen la qualitat de cada nucleòtid segons una escala[5].

A partir d'aquí comença la part computacional, per començar es fa un filtratge de les dades per tal de reduir el volum de dades, filtrant així les dades de menor qualitat que no poden ser considerades com a correctes. Tot seguit es comparen les seqüències de cada genoma i s'alineen les parts conflictives que poden desencadenar una possible mutació, a continuació s'identifiquen el tipus i posició de cada mutació i es presenten en un format estàndard que en permet la validació.

Un cop fet l'alineament també es poden guardar les dades en aquell punt en format BAM[6]. A més també es permet començar l'aplicació amb arxius d'aquest format, permeten d'aquesta manera continuar amb el flux de dades utilitzant dades que provinguin d'una execució anterior o inclús d'un altre programa.

1.3 Objectius

L'objectiu principal d'aquest projecte és presentar una implementació vàlida utilitzant de rerefons la idea de SMUFIN de contraposar les seqüències sanes i tumorals d'un pacient de la primera part del flux de dades de SMUFIN en concret, llegir els arxius fastq de forma efectiva, i amb la informació d'aquests arxius crear una estructura de tipus *Suffix Tree*[7], que ens permeti buscar-hi les mutacions. Aquesta estructura s'haurà de crear entre diferents nodes i per tant haurà d'estar correctament paral·lelitzada per tal d'aprofitar els màxims recursos dels que es disposi (sobretot pel què fa a memòria). A més intentarem en la mesura del possible reduir el nombre de llibreries utilitzades per no tenir un gran nombre de dependències externes.

També s'intentarà utilitzar aquesta estructura per donar una alternativa vàlida per el problema dels estudis poblacionals. En aquest apartat buscarem demostrar que l'alternativa que presentem es pot aplicar en aquest camp. Nosaltres utilitzarem un conjunt reduït de dades i en validarem el resultat i demostrarem que és escalable a nivell poblacional.

En ambdós casos el principal problema serà el gran ús de memòria i és per això que en els casos que haguem de prendre decisions entre reduir la memòria o reduir el temps d'execució, escollirem la primera.

---

[5] Per a més informació consultar: https://en.wikipedia.org/wiki/FASTQ_format
[6] Els fitxers .bam són fitxers binaris que normalment estan lligats a fitxers en format .sam que permeten mitjançant eines com SAMtools veure gràficament l'alineament fet i indexat amb tots els reads d'entrada.
[7] Més endavant veurem com és aquesta estructura.

1.4 Estat de l'art

1.4.1 Antecedents en mutacions somàtiques

L'augment que hi hagut darrerament pel que fa a la investigació i detecció de mutacions somàtiques ha provocat un creixement pel que fa a nombre de projectes relacionats amb aquest camp en tots els àmbits tant a nivell estatal, europeu i com a nivell mundial, provocant l'aparició de diferents aplicacions que són capaces d'analitzar genomes i detectar-ne mutacions somàtiques. La diferència que tenen aquestes aplicacions amb SMUFIN és que totes utilitzen sempre alguna eina per indexar els genomes amb l'anomenat Genoma de Referència, fet que els hi facilita la feina en el moment d'alinear els *reads* però que provoca l'augment de falsos positius degut a les diferències entre el genoma del pacient i el de referència.

Com es veu en l'article de SMUFIN[3], totes les aplicacions que hi estan mencionades, tenen uns resultats pel que fa a especificitat i sensibilitat menors a aquest, degut entre d'altres al problema que genera la indexació amb el genoma de referència. Veiem d'aquesta manera que SMUFIN actualment té una força avantatge respecte els seus competidors i per tant consolida la idea d'utilitzar l'estructura de *Suffix Tree*, ja que no tenim constància que cap dels seus competidors l'utilitzi. Programes detectors de mutacions somàtiques semblants a SMUFIN:

- *Pindel* [4]
- *BreakDancer* [5]
- *GenomeSTRIP* [6]
- *Genome Analysis Toolkit* (*GATK*) [7]
- *CREST [8]*
- *Delly[9]*
- *MuTect* [10]
- ...

Tot i tenir estratègies o pipelines diferents, en un punt del programa tots tenen en comú que acaben utilitzant alguna eina com són el MAQ (Mapping and Assembly Qualities) o el SSAHA2 (Sequence Search and Alignment by Hashing Algorithm) que els permeten alinear amb el genoma de referència.

Un dels projectes que té més similituds és un que s'està duent a terme en paral·lel, també en el BSC, pel grup d'investigació de David Carrera, en què es pretén crear una estructura amb els *reads* normals i tumorals. En aquest cas però utilitzant en lloc de totes les subseqüències els anomenats k-mers[8]. Aquesta estratègia és més ràpida i no té un consum tant alt de memòria però implica guardar menys informació en l'estructura final. En el fons es tracta de la mateixa idea que el Suf*fix Tree* però reduint-te la quantitat d'informació que aquest emmagatzema i sense connectar els diferents nodes.

1.4.2 Estat actual d'estudis poblacionals

Els estudis poblacionals o genètica de poblacions, és una branca de la genètica que intenta descriure la variació i distribució de la freqüència al·lèlica per explicar els fenòmens evolutius. Per això es defineix a una població com un grup d'individus de la mateixa espècie que estan

[8] Un k-mer és una subseqüència de mida k extreta d'una seqüència més gran. Una de les possibilitats per a l'anàlisi de seqüències d'ADN és generar totes les subseqüències de mida k per posteriorment recopilar dades de repeticions o divergències entre subseqüències.

aïllats, comparteixen el mateix hàbitat i es reprodueixen entre ells. Així doncs les poblacions al llarg del temps pateixen diverses mutacions, afectades per la selecció natural per exemple, i l'objectiu dels estudis poblacionals és la d'estudiar les similituds i diferències que es produeixen dins d'una mateixa població per poder arribar inclús a comparar diferents poblacions.

Pel que fa al nostre projecte, el que farem és consolidar la idea del *Suffix Tree* tot aplicant-lo també en aquest tipus d'estudis. Per la manca de temps, no farem un anàlisi ni una aplicació dedicada a aquest objectiu sinó que modificarem algunes parts de la nostra aplicació per a que sigui viable l'ús de quasi la mateixa eina. La mateixa essència del títol ens fa indicar que el programa ha de suportar un gran volum de dades ja que ara no estarem parlant d'un sol individu sinó de varis i que per tant posarem a prova l'escalabilitat de l'aplicació.

Com hem comentat anteriorment aquesta part de la genètica està en augment també gràcies a les millores en el camp de la informàtica en quant a volum de dades, i sobretot a la paral·lelització que permet reduir el temps d'execució significativament. No és d'estranyar doncs que el nombre d'aplicacions relacionades amb els estudis poblacionals no pari de créixer, aquí tenim una llista d'algunes d'elles:

- Churchill [11]
- PyPop [12]
- PopGenome [13]
- Stacks [14]
- Genome STRiP 1000g [15]
- …

Dia a dia la llista va en augment però de moment no tenim cap constància de cap altre projecte que hagi utilitzat una implementació de *Suffix Tree* o de *Suffix Array.*

1.4.3 Eines utilitzades: *MPI, Suffix-Tree, C++*

MPI

MPI (*Message Protocol Interface*) és un sistema d'enviaments de missatges estàndard i portable dissenyat per un grup d'investigadors tant d'empresa com acadèmics per a funcionar en un gran nombre d'ordinadors paral·lels. Està orientat per a sistemes amb memòria distribuïda. El seu estàndard defineix una sintaxis i una semàntica que són el nucli de les rutines de la llibreria a més és portable a varis llenguatges com Fortran, C++, C o Java, tot i que els més comuns són els dos primers. La gran avantatge respecte a altres llibreries d'enviament de missatges és que aquesta ha set implementada per a quasi tots els sistemes d'arquitectura de memòria distribuïda gràcies a la seva velocitat i portabilitat l'han fet ser un dels més utilitzats en sistemes d'aquest tipus de memòria.

Hem escollit aquesta tecnologia perquè és de les més utilitzades en el camp de memòria distribuïda, a més és compatible amb el llenguatge que utilitzem i ràpida en quant a velocitat. Una altra avantatge és que ja està instal·lat al *MareNostrum III,* en concret la versió Open MPI 1.8.1.

Una de les altres opcions que havíem considerat viable era l'ús de l'algoritme de MapReduce, un algoritme molt comú entre problemes de BigData[9], que es compon de dues parts, la primera la de *Mapping* i la segona la de *Reduce*, tal i com ho vam pensar la primera l'utilitzaríem per a construir l'arbre i la segona per la recerca de mutacions dins l'arbre. Finalment però vam desistir a provar-ho degut a que MPI ens semblava més bona opció sobretot perquè MapReduce necessita inicialitzar un rerefons, cosa que provoca una pèrdua inicial respecte MPI. S'han descrit de forma teòrica la utilització de *Map Reduce*[16] i tenim el cas pràctic de GATK (*Toolkit for Genome Analysis*). Externament a aquest projecte es possible que es provi de fer una implementació utilitzant el MapReduce.

Suffix-Tree

L'estructura que construïm és la *Suffix Tree* que ha estat desenvolupada en un projecte que s'ha realitzat en paral·lel amb aquest, és per això que cada cert temps hem anat rebent actualitzacions d'aquesta estructura pel que fa a la memòria utilitzada. Ja fa alguns anys que s'havia plantejat la idea d'utilitzar aquest tipus d'estructura en el camp de la bioinformàtica en concret per a seqüenciar genomes o proteïnes[17], d'aquesta en va sortir la idea del *Suffix Array* [18] una estructura semblant, més òptima pel que fa a memòria però que no ens permet emmagatzemar tota la informació que l'aplicació requereix, és per això que al final ens vam decantar pel *Suffix Tree*.

C++

C++ és un llenguatge de programació del que s'anomena nivell mitjà[10], dissenyat el 1980 per Bjarne Stroustrup, a partir del llenguatge C amb la intenció d'afegir-hi la manipulació d'objectes. Al ser de mitjà nivell ens ofereix un rendiment superior comparat amb altres llenguatges de programació com Java, Perl, Phyton etc. Està molt estès en el camp de la supercomputació (*HPC, High Perfomance Computer*) sobretot pel que fa el seu rendiment, la gran compatibilitat que té amb moltes aplicacions com són *MPI, OpenMP* etc. A més degut a que l'estudiant té un gran coneixement d'aquest llenguatge ha fet que sigui el llenguatge que hem decidit utilitzar. Com a eina de documentació s'utilitzarà el *Doxygen* que és una eina que ja ha set utilitzada al llarg del Grau per l'estudiant i que ens permetrà obtenir una documentació automàticament i en format tant *pdf* com *HTML*.

1.5 Competències

Les competències tècniques associades en un inici a aquest projecte són les següents:

- **CEC2.1:** Analitzar, avaluar, seleccionar i configurar plataformes hardware per al desenvolupament i l'execució d'aplicacions i serveis informàtics. [En profunditat]

[9] El *BigData,* és un concepte que fa referència a l'acumulació massiva de dades i a l'anàlisi d'aquestes, en el nostre cas, als procediments utilitzats per a identificar patrons recurrents.
[10] Una forma de diferenciar llenguatges és pel que se n'anomena nivell del llenguatge, entenent el nivell la manera en com s'expressa un algoritme en el llenguatge, sent nivell baix adequada a la capacitat cognitiva humana i nivell alt semblant a la capacitat executora de les màquines [19].

- **CEC2.2:** Programar considerant l'arquitectura hardware, tant en assemblador com en alt nivell. [En profunditat]

Aquestes dues competències s'han portat a terme durant la realització del projecte en el moment de la programació de l'aplicació i en idear l'algoritme general de l'aplicació i més en concret en dissenyar una estratègia de comunicació que aprofités al màxim tots els recursos i que tingués en consideració l'estructura del supercomputador on s'executava tot i tenir present la portabilitat de l'aplicació.

Competències genèriques del grau treballades durant el projecte:

En l'apartat de Comunicació Eficaç Oral i Escrita, amb els debats dintre del grup de treball i amb les reunions tant amb els codirectors com amb altres membres del grup s'ha treballat la competència següent:

G4: Comunicar de forma oral i escrita amb altres persones coneixements, procediments, resultats i idees. Participar en debats sobre temes propis de l'activitat de l'enginyer tècnic en informàtica.

En l'apartat de Treball en equip, degut a l'estructura de treball organitzada pels codirectors amb reunions setmanals en les que designàvem els objectius a assolir a una setmana vista:

G5: Ser capaç de treballar com a membre d'un equip, com a un membre més, ja sigui realitzant tasques de direcció, amb la finalitat de contribuir a desenvolupar projectes d'una manera pragmàtica i amb sentit de la responsabilitat; assumir compromisos tenint en compte els recursos disponibles.

En l'apartat d'Aprenentatge autònom, amb l'anàlisi que hem fet a posteriori de les execucions de l'aplicació i les millores que hem aplicat i que han millorat l'aplicació final:

G7:Detectar carències en el coneixement propi i superar-les mitjançant la reflexió crítica i l'elecció de la millor actuació per ampliar aquest coneixement. Capacitat per a l'aprenentatge de nous mètodes i tecnologies, i versatilitat per a adaptar-se a noves situacions

Pel que fa a les **competències tècniques** de l'especialitat d'Enginyeria de Computadors:

Degut a la morfologia del supercomputador *MareNostrum III* i la necessitat de que l'aplicació fos paral·lela i per tant amb la necessitat de comunicar diferents computadors:

CEC1: Dissenyar i construir sistemes digitals, incloent computadors, sistemes basats en microprocessadors i sistemes de comunicacions.

CEC2: Analitzar i avaluar arquitectures de computadors incloent plataformes paral·leles i distribuïdes, i desenvolupar i optimitzar software per a aquestes plataformes.

2 Planificació

El temps aproximat per a la durada d'aquest projecte és d'uns 9 mesos, període comprés des de febrer fins a octubre de 2015, estimant que l'estudiant dedicarà entre 5 i 6 hores diàries durant els dies d'entre setmana i 1 hora els dies de cap de setmana. A continuació es detalla la planificació del projecte, seguidament trobarem la planificació inicial i finalment es farà un comentari sobre els canvis que hi ha hagut. També es detallarà un pla d'acció utilitzat per a reduir els imprevistos i endarreriments que hi ha hagut durant el projecte.

Totes les tasques que es descriuen a continuació són realitzades pel desenvolupador del projecte.

2.1 Descripció de les tasques

Gestió del Projecte

En aquesta tasca s'hi inclou bàsicament la feina que es desenvolupa a l'assignatura de GEP. Consta de 7 entregables (subtasques), els quals tenen un temps de dedicació diferents. El total de dedicació a aquesta tasca són 75 hores. Els entregables són els següents:

- Abast del projecte (9.25 hores)
- Planificació temporal (8.25 hores)
- Gestió econòmica i sostenibilitat (9.25 hores)
- Presentació preliminar (6.25 hores)
- Contextualització i bibliografia (15.25 hores)
- Plec de condicions (8.5 hores)
- Presentació oral i document final (18.25 hores)

Els recursos necessaris són un ordinador, el *Microsoft Office*, l'aplicació *Gantter*, un visualitzador de PDF, una càmera, el Racó de la FIB, l'Atenea de la UPC, el *Dropbox*, el *Google Drive* i connexió a Internet.

Estudi previ a la implementació

Degut a la complexitat de la implementació que s'ha de dur a terme s'han d'estudiar i analitzar les diferents eines i models que s'utilitzaran.

Anàlisi del model i algoritme *Suffix Tree*

Analitzar l'algoritme de creació de *Suffix Tree* que serà necessari per tal d'entendre l'estructura, funcionament i avantatges dels *Suffix* Tree respecte als altres tipus d'implementacions que es podrien utilitzar, tot llegint articles que puguin ajudar en la tasca d'entendre'l.

Un cop acabada la tasca el realitzador del projecte serà capaç d'entendre i construir un *Suffix Tree*.

Els recursos necessaris són un ordinador, un visualitzador de PDF i connexió a Internet.

El temps destinat a aquesta subtasca és de 15 hores.

Anàlisi del model MPI

Aquesta tasca s'iniciarà un cop acabada la tasca anterior (Anàlisi del model i algoritme *Suffix Tree*).

Degut a que s'ha de dur a terme una implementació d'un algoritme utilitzant el model MPI es necessari l'anàlisi i estudi d'aquest model per a que sigui utilitzat correctament i eficientment. També caldrà buscar una implementació que pugui executar-se al *MareNostrum III* (a poder ser que ja hi estigui instal·lada). Aquesta implementació haurà de ser estudiada i instal·lada (també a l'ordinador personal) de manera que l'estudiant pugui realitzar la seva feina des de l'ordinador personal.

Els recursos necessaris són un ordinador, un visualitzador de PDF i connexió a Internet.

El temps d'aquesta subtasca és de 20 hores.

Familiarització amb l'entorn de treball i execució

Aquesta tasca s'iniciarà un cop acabada la tasca anterior (Anàlisi del model MPI).

Abans de començar a treballar amb la implementació serà necessari que l'estudiant es familiaritzi amb l'entorn de treball i sobretot en com s'executen les aplicacions al MareNostrum III. Per exemple es provarà la entrada i sortida de fitxers, com indicar el nombre de processadors a treballar en paral·lel, com utilitzar el gestor de versions, etc. També caldrà veure les línies de comandes amb les quals es comunica l'ordinador de treball amb el supercomputador des de l'usuari i contrasenya designat a les eines de comunicació.

D'aquesta manera l'estudiant serà capaç de poder provar la seva aplicació durant el desenvolupament d'aquesta i un cop acabat al *MareNostrum III* de manera ràpida i eficient.

Els recursos necessaris per aquesta tasca són: Un ordinador que pugui establir connexions remotes xifrades *SSH*(*Secure SHell*), accés a Internet i disposar d'un compte d'usuari al *MareNostrum III*.

El temps d'aquesta subtasca és de 10 hores tenint en compte que l'estudiant ja disposa de compte del *MareNostrum III* i que no hi ha cap manteniment previst que obstaculitzi el treball.

Implementació de l'aplicació utilitzant *MPI*

Un cop acabada la tasca anterior l'estudiant podrà començar a desenvolupar la seva implementació de l'algoritme de *Suffix Tree* sota la implementació de *MPI* escollida prèviament. Cal mencionar en aquesta tasca també l'aportació de Jordi Cardona, Friman Sánchez i David Torrents, que amb reunions mensuals han permès una ràpida evolució de l'aplicació amb les seves idees.

Per tal d'estructurar aquesta tasca, que és la més extensa, s'ha decidit dividir-la en quatre subtasques que detallen amb més profunditat la feina feta.

Lectura de fitxers fastq

La primera tasca de la implementació és la de programar un lector d'arxius *fastq* que sigui eficient i dinàmic, degut al format dels arxius, el fet que fos molt dinàmic (amb un element variable per l'usuari), i que s'executa moltes vegades ha fet que la complexitat d'aquesta tasca fos alterada i provoqués que la durada final fos més llarga de la que en un principi s'esperava, en total unes 120 hores.

Per a realitzar aquesta tasca hem necessitat els següents recursos: arxius de prova en format *fastq*, compilador *gcc*, un ordinador personal, un servidor de repositoris i un editor de text (*Sublime Text*).

Comunicació

Aquesta tasca és la que correspon a la comunicació, en concret de buscar una estratègia òptima de pas de missatges utilitzant MPI.

Per tant és l'encarregada de dissenyar l'esquema general de l'aplicació, tant com es llegeixen els fitxers d'entrada fins a com s'estructura el Suffix Tree, a més de l'estratègia també cal programar la implementació que per tant ha d'incloure la lectura de fitxers i la comunicació entre tots els nodes. S'han provat diferents formes de comunicar els nodes el que ha comportat un temps de dedicació elevat en aquest tasca, en concret, d'aproximadament 280 hores.

Els recursos necessaris han estat els següents: arxius de prova en format fastq, compilador de *MPI (*mpi*++)*, un ordinador que pugui establir connexions remotes xifrades *SSH*(*Secure SHell*), un servidor de repositoris, nodes del *MareNostrum III*, connexió a internet i un editor de text (*Sublime Text*).

Creació de l'arbre

El principal objectiu d'aquesta tasca és la d'afegir la creació de *Suffix Trees* dins l'aplicació actual fent-ho de forma eficient i en disposició de l'estratègia escollida en la tasca anterior, per això caldrà buscar una estructura eficient capaç d'emmagatzemar els *Suffix Trees*. En la realització de la tasca es faran les proves amb un genoma *"In silico"* (expressió llatina que significa fet per ordinador), en concret amb el cromosoma 20 d'aquest, tenint així un joc de proves més gran que els duts a terme fins ara. La durada d'aquesta tasca ha set de 95 hores.

Per aquesta tasca s'han requerit els següents recursos: Arxius d'entrada del cromosoma 20 del genoma "*In silico"*, ordinador personal capaç d'establir connexions remotes xifrades *SSH*, un servidor de repositoris, nodes del *MareNostrum III*, connexió a internet, un compilador de *MPI* i un editor de text.

Validació i Obtenció de Resultats

Aquesta és la última tasca dins el bloc de la Implementació i és la que ha de validar tot el procés fins aquest punt, començant per elaborar i programar l'estratègia de validació i idear també una manera d'obtenir els resultats i crear-la. La durada d'aquest tasca és de 191 hores.

Els recursos que s'han requerit han estat els següents: Arxius d'entrada del cromosoma 20 del genoma "*In silico*", ordinador personal capaç d'establir connexions remotes xifrades SSH, nodes del MareNostrum III, connexió a internet, un compilador de MPI i un editor de text.

Anàlisi i Millores

Arribats a aquest punt on ja tenim una primera versió funcional, ens centrarem en analitzar el rendiment de l'aplicació i en buscar i aplicar millores tant en l'àmbit de memòria com en el de temps d'execució ja sigui per la comunicació o per alteracions en l'algoritme general de l'aplicació.

En aquesta tasca necessitarem eines per avaluar el rendiment com son el *Valgrind*, el *Paraver* i *Extrae*, a més d'un ordinador personal capaç d'establir connexions remotes xifrades SSH, nodes del MareNostrum III, connexió a internet, un compilador de MPI i un editor de text. La durada d'aquest tasca és de 60 hores.

Redacció Memòria

En aquesta tasca es tractarà de desglossar tot el què s'ha aprés durant el projecte i plasmar-ho a la memòria, per precaució (en quant a limitació de temps) la redacció es va iniciar a mitjans de juny de 2015, tot i que finalitzarà més tard que la tasca anterior per tal d'introduir tot el procés.

Per aquesta tasca s'han necessitat els següents recursos: Un ordinador personal, un editor de text (*Microsoft Office Word*), connexió a internet, l'aplicació *Gantter* (generador de diagrames de Gantt) i el *Dropbox*.

Presentació TFG

L'objectiu d'aquesta tasca és el d'elaborar la presentació final d'aquest projecte que s'utilitzarà durant la defensa d'aquest, per tal d'encabir tot el projecte es realitzarà un cop acabada la redacció. El temps aproximat és d'unes 20 hores.

Per a realitzar aquesta tasca s'ha necessitat un ordinador personal amb connexió a internet i una aplicació per preparar presentacions (*Microsoft Office PowerPoint*).

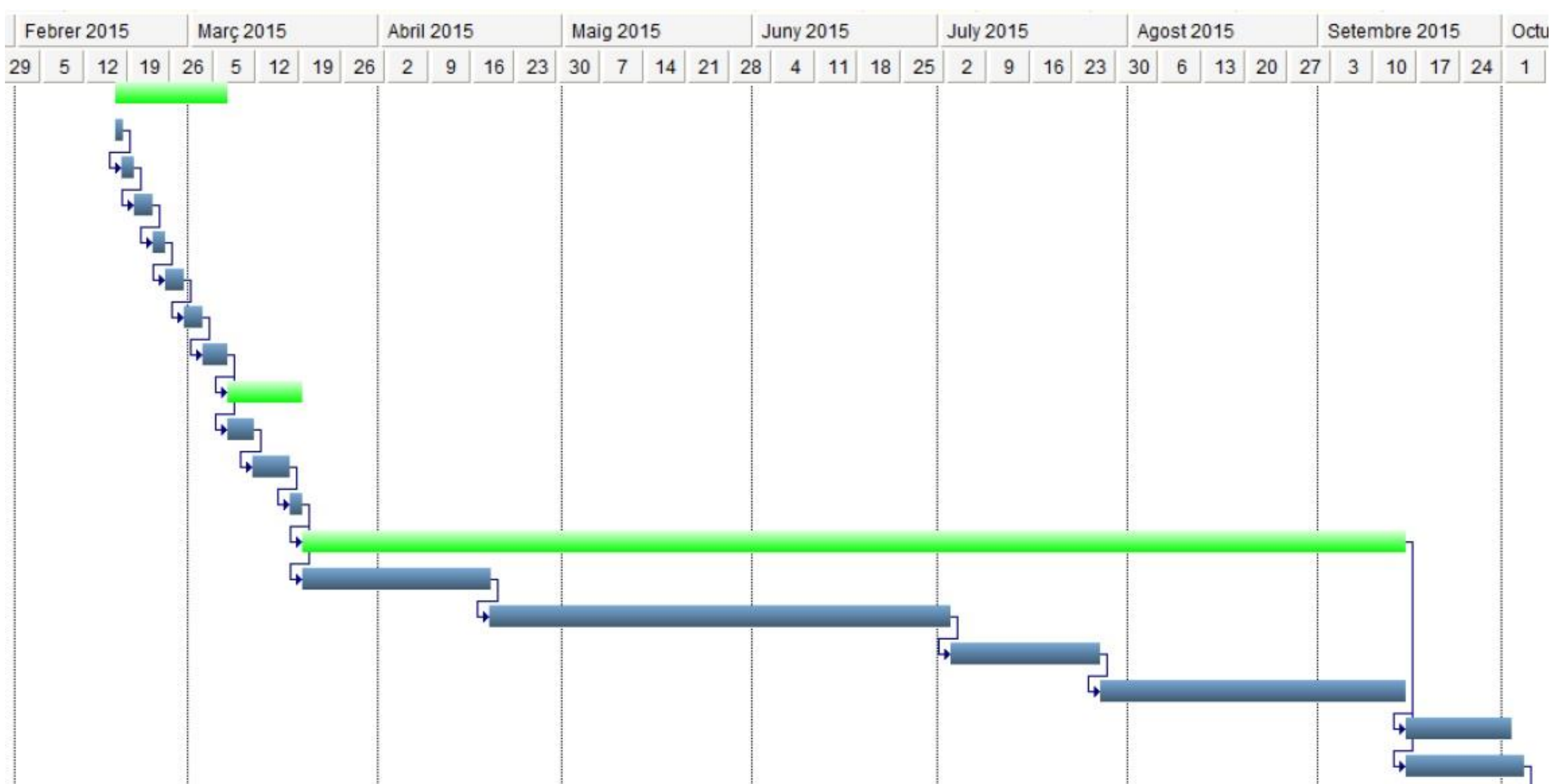
2.2 Explicitació de l'ordre lògic de les tasques

Tasca/Subtasca	Tasca/Subtasca Predecessora
Abast del projecte	-
Planificació temporal	Abast del projecte
Gestió econòmica i sostenibilitat	Planificació temporal
Presentació preliminar	Gestió econòmica i sostenibilitat
Contextualització i bibliografia	Presentació preliminar
Plec de condicions	Contextualització i bibliografia
Presentació oral i document final	Plec de condicions
Anàlisi del model i algoritme *Suffix Tree*	Presentació oral i document final
Anàlisi del model *MPI*	Anàlisi del model i algoritme *Suffix Tree*
Familiarització amb l'entorn de treball i execució	Anàlisi del model *MPI*
Implementació	Familiarització amb l'entorn de treball i execució
Anàlisi i millores	Implementació
Redacció memòria	Implementació
Presentació	Redacció memòria

Taula 1: Explicitació de l'ordre de les Tasques/Subtasques

2.3 Diagrama de Gantt

A continuació veiem el diagrama de Gantt de la planificació final on les tasques en verd il·lustren les tasques que engloben altres tasques:



Il·lustració 2: Diagrama de Gantt de la Planificació final

L'ordre de les tasques anteriors és la següent, a més en la imatge s'il·lustren les durades, la data inicial i final i els predecessors de cada tasca:

		Nom	Durada	Inici	Fi	Predecessors
1		Gestió del Projecte	75h	16/02/2015	06/03/2015	
2		Abast del Projecte	9.25h	16/02/2015	17/02/2015	
3		Planificació Temporal	8.25h	17/02/2015	19/02/2015	2
4		Gestió Econòmica i Sostenibilitat	9.25h	19/02/2015	22/02/2015	3
5		Presentació Preliminar	6.25h	22/02/2015	24/02/2015	4
6		Contextualització i Bibliografia	15.25h	24/02/2015	27/02/2015	5
7		Plec de Condicions	8.5h	27/02/2015	02/03/2015	6
8		Presentació oral i document Final	18.25h	02/03/2015	06/03/2015	7
9		Estudi previ a la implementació	45h	06/03/2015	18/03/2015	8
10		Anàlisi del model i algoritme Suffix Tree	15h	06/03/2015	10/03/2015	8
11		Anàlisi del model MPI	20h	10/03/2015	16/03/2015	10
12		Familiarització amb l'entorn de treball i execució	10h	16/03/2015	18/03/2015	11
13		Implementació	686h	18/03/2015	11/09/2015	12
14		Lectura de fitxers fastq	120h	18/03/2015	17/04/2015	12
15		Comunicació	280h	17/04/2015	30/06/2015	14
16		Creació de l'arbre	95h	30/06/2015	24/07/2015	15
17		Validació i Obtenció de Resultats	191h	24/07/2015	11/09/2015	16
18		Anàlisi i Millores	60h	11/09/2015	28/09/2015	13
19		Redacció de la Memòria	70h	11/09/2015	30/09/2015	13
20		Presentació	20h	30/09/2015	05/10/2015	19

Il·lustració 3: Característiques de les tasques

2.4 Anàlisi de les modificacions en la planificació inicial

Durant la realització s'han produït desviacions respecte a la planificació inicial, la més important de totes ha set que inicialment vàrem preveure de fer l'aplicació utilitzant el paradigma de *MapReduce* però al començar a investigar en el tema ens va sorgir la idea d'utilitzar *MPI* que ha set l'elecció final del projecte degut a que les complicacions que produïa la inicialització de *MapReduce* ens semblaven que contrarestaven forces respecte a *MPI* que no requereix de cap inicialització complexa.

El fet és que la planificació inicial com hem dit es va basar en *MapReduce* que finalment no hem utilitzat, però que en un futur ens agradaria donar-li una oportunitat val a dir, per tant les dues planificacions divergeixen bastant en conceptes però en el fons es tracten de dues planificacions semblants però amb tecnologies diferents.

Una altra desviació important és que inicialment volíem incloure la nostra aplicació dins el flux de dades de *SMUFIN* però de moment no serà així ja que l'objectiu ha canviat a crear una nova versió de *SMUFIN* i per tant aquest part també divergeix de la planificació inicial ja que ara no es requerirà ni fer l'estudi ni moure l'aplicació dins *SMUFIN.*

Una de les desviacions (petita) que més hem patit ha set produïda degut a la demanda d'execucions al MareNostrum III. Degut a la demanda feta per a cada execució de nombre de processadors, ens hem trobat que en molts casos les execucions tardaven un dia en executar-se i per tant mentre es posava una execució a la cua d'execucions s'havia d'aprofitar el temps en que aquesta no s'executava en altres tasques.

Pel què fa a les tasques de familiarització amb l'entorn de treball, les d'estudi dels models (canvi de *MapReduce* a *MPI*) utilitzats i l'etapa final de redacció i presentació del treball no han patit cap canvi.

2.5 Valoració d'alternatives i pla d'acció

Gràcies a l'aplicació de la metodologia *Scrum*, que és molt dinàmic, ens ha permès que tot i que es produïssin aquests canvis abans mencionats no s'hagi vist alterat el treball de fons amb els mencionats *Sprints*[11] i les reunions setmanals amb un dels codirectors, ja que cada setmana es marcaven un objectiu per la setmana següent i per tant hem anat modelant el treball a fer de manera seqüencial i regular. El pla d'acció que ha sigut necessari és el següent:

Si en una de les Reunions setmanals es detecta una desviació de la planificació caldrà:

- Si una tasca s'ha finalitzat abans del que s'havia establert, no hi ha cap problema, s'avançarà a la següent tasca.
- Si una tasca dura més del planificat, s'allarga i es comença la següent més tard. Si l'endarreriment és molt significatiu caldrà que l'estudiant dediqui més hores i que intenti tenir una solució menys efectiva en termes de temps i consum de recursos de la màquina ja que en una tasca posterior ja es mirarà d'optimitzar.

[11] *L'Sprint* és el període en què es realitza l'increment del producte.

Degut a que els recursos entre les tasques són molt semblants, aquestes desviacions no haurien de produir cap problema en aquest apartat.

3 Pressupost

3.1 Identificació i estimació dels costos

A continuació es detallaran els diferents elements a considerar en l'estimació del pressupost del projecte. Primer es calcularan els costos directes, seguidament els indirectes i per acabar es posarà tot en context i es calcularan contingències i imprevistos.

Costos Directes

Es farà un càlcul estimat per a cada tasca programada. En moltes de les tasques s'utilitzen recursos gratuïts i que no generen cap tipus de cost, en aquests casos només s'esmentarà els recursos mencionats. En els recursos que generin costos directes es calcularan els preus unitaris, la seva vida útil, la seva amortització estimada i per acabar el preu final en funció dels paràmetres anteriors.

Es calcula una mitjana de 253 dies laborables per any en funció dels pròxims 4 anys que s'utilitzarà pel càlcul de l'amortització i es calcularà en base a 8 hores laborables.

El cost del supercomputador MareNostrum III en les tasques que s'utilitza es posarà el preu unitari però no es calcularà l'amortització ni el preu final degut a que és un supercomputador d'ús públic.

Els recursos humans del projecte només els constitueixen un estudiant d'enginyeria Informàtica especialitzat en computadors, no es tindran en compte altres actors. El cost per hora és de 20 euros.

Gestió del Projecte

Els recursos de cost zero que s'utilitzen en aquesta tasca són: l'aplicació *Gantter*, un visualitzador de PDF, el Racó de la FIB, l'Atenea de la UPC, el Dropbox, el Google Drive.

La resta de recursos són els següents:

	Unitats	Preu Unitari (€)	Vida Útil	Amortització estimada	Preu (€)
Ordinador Portàtil (inclou ratolí)	1	1000	4	9.26	9.26
Microsoft Office	1	100	3	1.24	1.24
Càmera	1	120	4	1.11	1.11
Microsoft Office Project	1	460	3	5.68	5.68
Recursos humans (Enginyer Informàtic)	75	20	-	-	1500
Total					1517.28

Taula 2: Costos directes GEP

Estudi previ a la implementació

L'estudi previ conté les 3 subtasques següents:

- Anàlisi del model i algoritme *Suffix Tree* *(taula2)*
- Anàlisi del model MPI *(taula 3)*
- Familiarització amb l'entorn de treball i execució*(taula 4)*

Totes tres tasques són molt semblants amb el context de que són tasques d'estudi per a l'estudiant i que per tant no tenen costos molt significatius, les tres comparteixen el següent recurs gratuït: visualitzador de PDF.

Els recursos que comporten costos directes es detallen a continuació:

Anàlisi del model i algoritme *Suffix Tree*	Unitats	Preu Unitari (€)	Vida Útil	Amortització estimada	Preu (€)
Ordinador Portàtil (inclou ratolí)	1	1000	4	1.85	1.85
Recursos humans (Enginyer Informàtic)	15	20	-	-	300
Total					301.85

Taula 3: Costos directes Anàlisi del model i algoritme Suffix Tree

Anàlisi del model *MPI*	Unitats	Preu Unitari (€)	Vida Útil	Amortització estimada	Preu (€)
Ordinador Portàtil (inclou ratolí)	1	1000	4	1.85	1.85
Recursos humans (Enginyer Informàtic)	20	20	-	-	400
Total					401.85

Taula 4:Costos directes Anàlisi del model MPI

Familiarització amb l'entorn de treball i execució	Unitats	Preu Unitari (€)	Vida Útil	Amortització estimada	Preu (€)
Ordinador Portàtil (inclou ratolí)	1	1000	4	1.24	1.24
MareNostrum III	1	22700000	3	-	-
Recursos humans (Enginyer Informàtic)	10	20	-	-	200
Total					201.24

Taula 5:Costos directes Familiarització amb l'entorn de treball i execució

Implementació de l'aplicació utilitzant MPI

Els recursos sense costos d'aquesta tasca són un compilador, el servidor de repositoris (que s'utilitzarà un de gratuït), un compte al MareNostrum III, un editor de text gratuït com el Sublima Text o l'Emacs i un comparador d'arxius gratuït com el *diff*.

Degut a la durada i la dificultat d'aquesta tasca és possible que puguin sorgir imprevistos que comportin que la durada es vegi augmentada i en conseqüència els costos, com per exemple que l'estudiant no aconsegueixi fer que l'algoritme funcioni correctament per alguns casos o que es tardi més del planificat en la comprovació dels resultats de l'algoritme. En tot es pot preveure un 10% de risc en aquesta tasca, i aquest mateix percentatge serà calculat a continuació i mostrat a la següent taula.

	Unitats	Preu Unitari (€)	Vida Útil	Amortització estimada	Preu (€)
Ordinador Portàtil (inclou ratolí)	1	1000	4	32.11	32.11
MareNostrum III	1	22700000	3	-	-
Recursos humans (Enginyer Informàtic)	686	20	-	-	13720
Total					13752.11
Imprevistos	-	523.21(10%)			1375.21
Total + Imprevistos					15127.32

Taula 6: Costos directes Implementació de l'aplicació utilitzant MPI

Anàlisi i millores

Els recursos amb cost zero d'aquesta tasca són els següents: un compilador, un servidor de repositoris, un compte al MareNostrum III, un editor de text gratuït com el Sublime Text o l'Emacs, jocs de proves (proporcionades pel BSC i de descàrrega gratuïta), eines gratuïtes que analitzen traces d'un programa com el *Tareador* i una eina per visualitzar aquestes traces com el *Paraver* i *Extrae, Doxygen* per tal de generar la documentació del codi (gratuït) i un comparador d'arxius gratuït com el *diff* o el *cmp*.

No es preveuen riscos en aquesta tasca degut a que encara no en sabem l'abast.

	Unitats	Preu Unitari (€)	Vida Útil	Amortització estimada	Preu (€)
Ordinador Portàtil (inclou ratolí)	1	1000	4	7.35	7.35
MareNostrum III	1	22700000	3	-	-
Recursos humans (Enginyer Informàtic)	60	20	-	-	1200
Total					1207.35

Taula 7: Costos directes Anàlisi i millores

Redacció de la memòria

Aquesta tasca tindrà els següents recursos a cost zero: un visualitzador de PDF, el Racó de la FIB, el Dropbox i l'editor de text Microsoft Office.

	Unitats	Preu Unitari (€)	Vida Útil	Amortització estimada	Preu (€)
Ordinador Portàtil (inclou ratolí)	1	1000	4	6.18	6.18
Microsoft Office	1	100	3	0.82	0.82
Recursos humans (Enginyer Informàtic)	70	20	-	-	1400
Total					1407

Taula 8: Costos directes Redacció de la memòria

Presentació

Per acabar l'última tasca tindrà els següents recursos a cost zero: un visualitzador de PDF, el Racó de la FIB i el Dropbox.

	Unitats	Preu Unitari (€)	Vida Útil	Amortització estimada	Preu (€)
Ordinador Portàtil (inclou ratolí)	1	1000	4	6.18	2.18
Microsoft Office	1	100	3	0.82	0.82
Recursos humans (Enginyer Informàtic)	20	20	-	-	400
Total					403

Taula 9: Costos directes Presentació

Costos Indirectes

Els costos indirectes d'aquest projecte són comuns en totes les tasques i es calcularan per a totes les tasques a la vegada, són els següents:

	Unitats	Preu Unitari (€/h)	Vida Útil	Amortització estimada	Preu (€)
Llum	956	0.05	-	-	47.8
Quota ADSL	956	0.09	-	-	86.04
Total					133.84

Taula 10: Costos indirectes de tot el projecte

Pressupost Total del projecte

	Unitats	Preu Unitari (€/h)	Vida Útil	Amortització estimada	Preu (€)
Tasca1(directes)					1517.28
Tasca2(directes)					301.85
Tasca3(directes)					401.85
Tasca4(directes)					201.24
Tasca5(directes)					15127.32
Tasca6(directes)					1207.35
Tasca7(directes)					1407
Tasca8(directes)					403
Indirectes					133.84
Total acumulat					20700.73
Contingència		5% Total acumulat			1035.04
Total sense IVA					21735.77
Total amb IVA (21%)					26300.28

Taula 11: Pressupost Total del projecte

El factor de risc es calcula en funció de la dificultat i les hores destinades a la tasca, en moltes d'elles és zero degut a que la tasca en sí no té molta dificultat.

Per altra banda, s'ha calculat el percentatge de la contingència. Per aquest projecte s'ha estimat un 5%, ja que el pressupost que està a continuació està molt detallat i que molts dels recursos usats no tenen cap cost.

Finalment s'aplica l'IVA (21%) sobre el cost acumulat del pressupost.

4 Cos del treball

4.1 Flux de dades de l'aplicació

A continuació es detallarà tot el flux de dades de la nostra aplicació, tot i que primer farem una explicació de fons de l'estructura sobre la qual es construeix l'aplicació tot i que no arribarem fins a un nivell de detall molt fi degut a que aquesta estructura és externa al treball pel que fa a implementació i rendiment.

4.1.1 *Suffix Tree*

El principal concepte d'aquesta nova versió és la utilització de l'estructura *Suffix Tree*, que es crearà a mida, amb només els requeriments d'aquest projecte, en un Treball de fi de Grau que es realitza en paral·lel, dins el mateix interval de temps que aquest, i que es dut a terme per Jordi Cardona, i a més la no dependència de llibreries externes, amb la mesura del possible.

El *Suffix Tree* és una estructura que com el seu nom indica, és en forma d'arbre. L'estructura es basa en el concepte dels sufixes, que són subseqüències que es poden generar d'una seqüència més llarga. Posem un exemple:

Si tenim la següent seqüència[12]: ACGTGA

Podem extreure vàries subseqüències, que ordenant-les de manera decreixent per la seva mida serien les següents:

1. ACGTGA (la seqüència pròpiament)
2. CGTGA
3. GTGA
4. TGA
5. GA
6. A

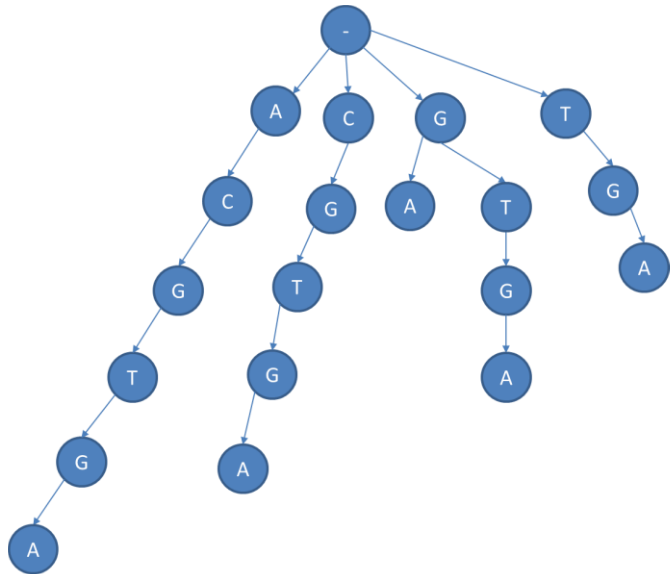
En algunes implementacions estudiades s'afegeix una setena subseqüència amb el símbol '$', que indica el final de seqüència, a més, aquest símbol es posa al final de totes les altres subseqüències. Nosaltres no ho hem considerat oportú en el nostre treball degut a que suposa entre altres coses emmagatzemar un caràcter més per a cada subseqüència o sigui incrementar l'ús de memòria per a cada node.

Un cop tenim les subseqüències o sufixes, podem construir un arbre, en el nostre cas, quaternari[13], suposant que cada node és un nucleòtid i que l'arrel la simbolitzem amb un signe negatiu '-, 'aquest seria el de l'exemple:

---

[12] Hem utilitzat una seqüència de nucleòtids tot i que es poden formar subseqüències de qualsevol tipus de seqüències, tan de nombres, lletres etc.

[13] Un arbre quaternari, és aquell què cada node només pot tenir com a màxim 4 fills, degut a que treballem amb nucleòtids només tenim quatre caràcters (A, C, G, T) i per tant només quatre possibles fills.



Il·lustració 4: Arbre de sufixes de la seqüència ACGTGA

L'arbre utilitzat en el projecte difereix del mostrat en matisos però en el fons és tracta de la mateixa idea. El motiu pel que difereix és degut a que com hem comentat l'arbre es farà a "mida", tenint en compte els aspectes que necessitem, en un altre projecte que es desenvolupa en paral·lel i de fet serà l'únic codi que nosaltres aprofitarem. Gràcies a aquestes optimitzacions en l'estructura reduirem el cost de memòria en l'arbre i per tant ens ajudaran a arribar al nostre objectiu de fer més portable la nostra aplicació.

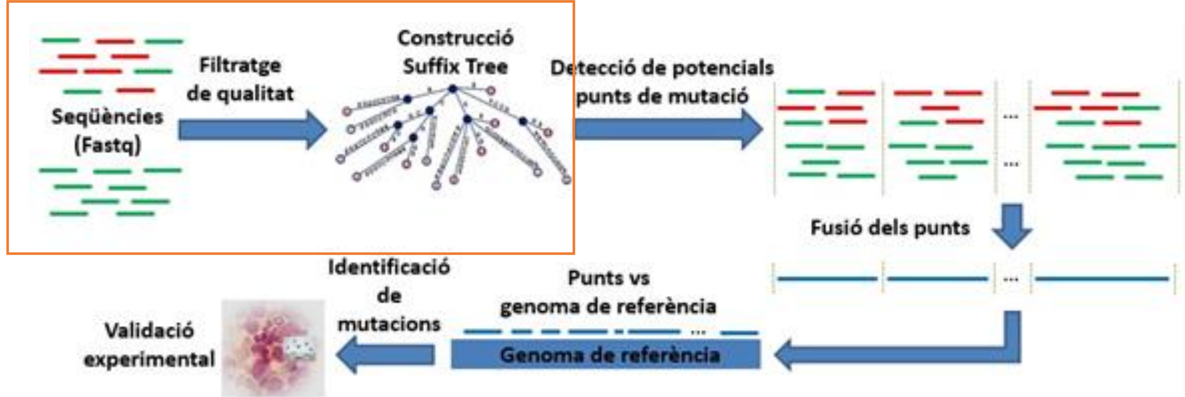
A dins de cada node hi hem afegit dos comptadors, un per normal i l'altre per tumoral, que ens indiquen el nombre de sufixes inserits en aquell punt. D'aquesta manera buscar dins l'arbre en serà molt més còmode ja que només mirant aquest comptadors ja podrem prendre decisions alhora de buscar possibles mutacions, per exemple podrem veure amb facilitat el punt on difereixen el sufixes sans i tumorals, tot comparant els comptadors d'un node "pare[14] " amb els dels seus "fills". Aquesta part però la veurem més endavant ja que és la part final del flux de dades, on busquem dins l'arbre els possibles punts de mutació.

Un altre matís important és en quin ordre inserim els *reads* dins l'arbre. La nostra idea és primer inserir tot els *reads* normals. Així una de les millores que podem incorporar és que quan inserim sufixes tumorals, en el moment en què divergim dels normals i per tant creem una branca nova o inclús un subarbre nou, no és necessari que continuem inserint ja que sabem que en aquells punts no hi ha cap sufix normal i per tant que no podem trobar cap divergència més endavant. Això ens permet reduir la càrrega de la inserció tumoral tant en memòria com en temps com veurem més endavant.

[14] Anomenem node "pare" a un node qualsevol en relació als nodes que d'ell sorgeixen, que definim com a fills.

4.1.2 Flux de dades

Pel que fa al flux de dades que seguirà la nostra aplicació és el següent, tot i que és un esquema general, aquest projecte només engloba la primera part, fins a "Detecció de potencials punts de mutació":



Il·lustració 5: Flux de dades de la nostra aplicació

L'aplicació començarà llegint els arxius *fastq* d'entrada, tan els normals (sans) com els tumorals, per a cada *read* (seqüència) es validarà que passa un control de qualitat i es filtraran els *reads* que no arribin al llindar establert. Un cop fet el filtratge inicial es començarà amb la construcció del *Suffix Tree*, primer s'hi insereixen els sufixes sans i un cop acabats aquests els tumorals. Construït l'arbre, ja podrem buscar possibles punts de mutació, per aquest apartat recorrerem tot l'arbre i analitzarem els comptadors abans mencionats.

Tot i que el flux de dades encara no s'haurà acabat, per al nostre projecte considerarem només aquesta part, en un futur es podrà continuar el flux de dades tot alineant els punts de mutació trobats, identificar definitivament els tipus de mutació trobats i exportar els resultats en un format vàlid.

4.2 Creant la 1a versió

4.2.1 Algoritme general de l'aplicació

L'algoritme general no és molt complex, donat uns arxius d'entrada (en format *fastq*) hem d'aconseguir construir un *Suffix Tree* entre varis nodes ja que el volum d'aquest arbre fa que sigui inviable d'utilitzar un sol node en termes de memòria requerida, i un cop construït necessitem buscar punts de possibles mutació dins l'arbre. El fet de tenir-lo entre varis nodes ens permet explotar la paral·lelització i per tant l'optimització tant de la construcció com de les posteriors cerques.

Aquest projecte està emmarcat dins un context més gran format per un grup de treball plural i que comporta que tinguem dependències amb altres parts (com l'estructura de l'arbre). En concret de l'algoritme general comentat, l'objectiu d'aquest projecte és realitzar tota la part

externa del *Suffix Tree*, sent així la lectura dels arxius, l'arribada de les dades d'entrada a tots els nodes perquè construeixin el *Suffix Tree* de manera equitativa en termes de memòria, i la validació de que l'estructura es generi correctament. En altres paraules la part de paral·lelització en quan a memòria distribuïda.

Altres tasques de l'aplicació que no es consideren part d'aquest projecte són per exemple: creació d'un *Suffix Tree* amb els elements necessaris, que es pugui comprimir, que la construcció també sigui paral·lela a nivell de *thread* (flux), poder-lo fer persistent i per tant poder emmagatzemar-lo en disc etc.

Per tant dins d'aquest projecte necessitarem pensar en com fer la lectura dels arxius d'entrada eficientment tenint en compte que aquests es troben repartits, també ens haurem de plantejar de quina manera distribuïm el *Suffix Tree* entre els diferents nodes ja que no és viable en termes de memòria utilitzar un sol node. Altres problemes que intentarem solucionar seran per exemple com recopilarem tots els resultats o com buscarem un sufix qualsevol dins l'arbre que està repartit entre els nodes...

A continuació veurem tot el flux de dades de l'aplicació i la manera en com s'han resolt els problemes amb els que hem anat topant.

4.2.2 Estructura de l'aplicació

Lectura de Fitxers

Per a llegir els fitxers d'entrada, hem creat quatre paràmetres anomenats --x_fastq_y (on "x" pot ser normal o tumor i "y" 1 o 2), seguidament hem de posar un arxiu que contingui les rutes dels arxius d'entrada ja que és el més comú fer-ho d'aquesta manera. El fet de separar en 1 i 2 tan els normals com els tumorals fa referència a la manera en què es seqüencia l'ADN.

Per tal de seqüenciar l'ADN s'agafa una "tira" d'aquest i es llegeixen els nucleòtids des dels extrems de la "tira". Imaginem-nos que aquesta tira és una corda posem de 5 metres, el procés de seqüenciació el que fa es tallar la corda en tres trossos, posem per exemple que talla 1 metre de les bandes el què ens fa obtenir dos trossos d'un metre i un de tres metres. Els dos trossos d'un metre són els que nosaltres anomenem *reads*, mentre que el restant no s'utilitza.

D'aquesta manera els dos trossos (o *reads*) que llegim tenen una proximitat relativa (acostuma a ser d'uns 500 nucleòtids aprox.) que ens pot servir en un futur i per tant ens és útil de guardar-ne la relació. Aquesta relació s'anomena *pair-ending* (parelles del final) i la forma de guardar la relació és que en lloc de crear un únic *fastq* amb tots els *reads* se'n creen dos, de manera que s'ordenen amb els *pair-end* això vol dir que el primer *read* de l'1 correspon al *pair-end* del primer read del 2. Per mantenir aquesta informació en la nostra aplicació el que fem és guardar els *ids* parells per un *pair-end* i els senars pels altres, ja que ens pot servir en la part de l'alineament. Per exemple si tenim els dos fitxers següents:

```
HW78768
ACGATCAGCTACGTAGTGTAGTGCAT
GC
+
JKBKBK&%·$UKKNO*LKVUVJUKYJVY
U&
HW78763
ACGATCAGCTACGTAGTGTAGTGCAT
GC
+
JKBKBK&%·$UKKNO*LKVUVJUKYJVY
U&
HW781268111
ACGATCAGCTACGTAGTGTAGTGCAT
GC
+
JKBKBK&%·$UKKNO*LKVUVJUKYJVY
U&
HW787148
ACGATCAGCTACGTAGTGTAGTGCAT
GC
```

```
HW787132
ACGATCAGCTACGTAGTGTAGTGCAT
GC
+
JKBKBK&%·$UKKNO*LKVUVJUKYJVY
U&
HW7151
ACGATCAGCTACGTAGTGTAGTGCAT
GC
+
JKBKBK&%·$UKKNO*LKVUVJUKYJVY
U&
HW781128111
ACGATCAGCTACGTAGTGTAGTGCAT
GC
+
JKBKBK&%·$UKKNO*LKVUVJUKYJVY
U&
GH788
ACGATCAGCTACGTAGTGTAGTGCAT
GC
```

Il·lustració 6: Exemple de dos arxius pair-end, en format fastq

Els *reads* del *fastq1* tindrien seguint l'arxiu, els *ids* següents: 0,2,4,6. Mentre que els del *fastq2* tindrien: 1,3,5,7. D'aquesta manera sense tenir cap tipus de pèrdua mantenim la relació de *pair-ending* en la nostra aplicació. Encara que un dels *reads* no passi els filtres de qualitat se'n reserva el seu *id* per mantenir la relació ja que un *pair-end* passi el filtre de qualitat no implica que el seu "company" també el passi i sinó ho féssim així estaríem establint les relacions malament.

Continuant amb la lectura el que fem ara és obrir els arxius (que contenent una llista de rutes als *fastq)* i ens guardem el contingut d'aquets arxius. Seguidament fem la lectura (ara ja si, dels *fastq*) llegint una quantitat de bytes, que també es pot modificar amb el paràmetre --*readblock*, i que carrega aquest bytes a un *buffer,* o sigui un buffer de lectura. Després analitzem aquest buffer per a aconseguir llegir els *reads* i les qualitats d'aquests. Un cop llegit un *read* ha de passar el filtre de qualitat, si el passa llavors es copia el *read* i l'identificador dins el *buffer* d'enviament.

Aquest procés es repeteix n vegades fins que s'ha llegit tot l'arxiu d'aquesta manera aconseguim llegir l'arxiu molt més ràpidament que si llegíssim línia a línia ja que importem més dades de cop del disc reduint doncs l'impacte del sistema operatiu i les crides per a llegir del disc (latència de disc).

M'agradaria comentar que aquesta part ens va portar molts problemes degut a que com que el paràmetre del *readblock* és configurable, que les 4 línies que conformen un *read* no sempre tenen la mateixa mida en bytes i que s'executa moltes vegades ja que els arxius són molt grans, calia controlar tots els possibles cassos que es poguessin produir i això ens va portar més feina de la que realment havíem pensat inicialment.

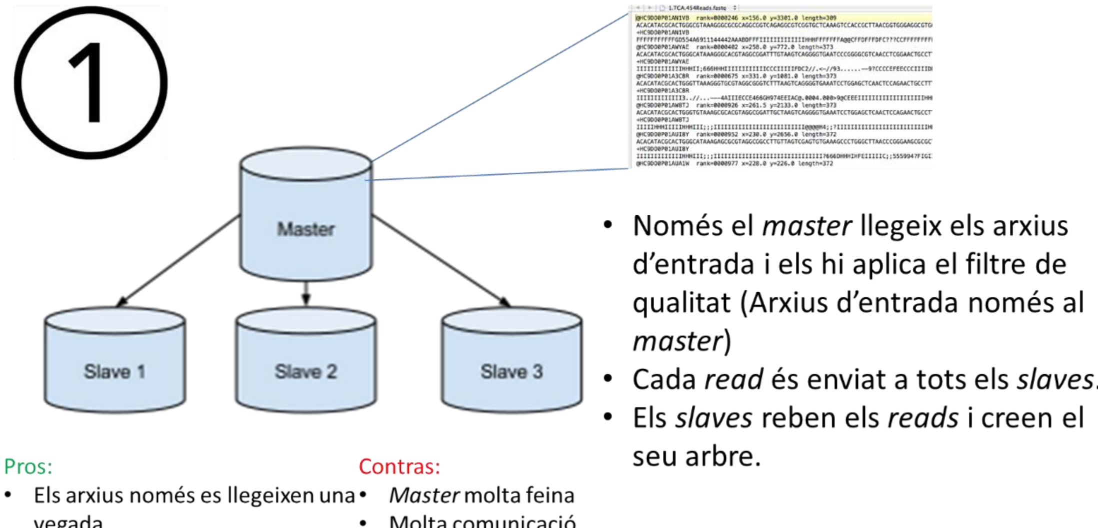
Comunicació

Tot partint de l'estructura que volem crear el *Suffix Tree* i tenint en compte altres factors com els tipus d'arxius d'entrada que s'utilitzen en aquest camp, hem de dissenyar com es gestionaran els recursos dels que disposem. Aquí val destacar alguns factors: com estan distribuïts els arxius d'entrada, l'ample de banda del que disposem entre els diferents nodes....

Inicialment vàrem pensar en tres estratègies diferents pel què fa la lectura dels arxius d'entrada:

Estratègia 1

La 1a d'elles és la que es mostra a continuació:



Il·lustració 7: Estratègia Inicial 1

En aquesta estratègia definim a un node com a *master* que és l'encarregat de llegir tots els arxius d'entrada. A mesura que els va llegint fa un primer filtratge que correspon a comprovar uns llindars de qualitat per a cada *read* i que si no els supera ja no l'envia als altres nodes. D'aquesta manera la informació transmesa es redueix ja que a part de que no enviem *reads* innecessaris tampoc enviem la qualitat ni altres dades secundàries que hi ha dins els arxius ja que els filtrem un sola vegada, només envia un llistat de *reads* amb el seu *id* corresponent. Per a millorar l'eficiència de l'enviament s'envien a un *buffer* i quan aquest està ple s'envia tot sencer. Cal mencionar també que el node *master* no construeix cap part de l'arbre i que per tant en un dels nodes no s'aprofitarà tot el potencial de memòria disponible.

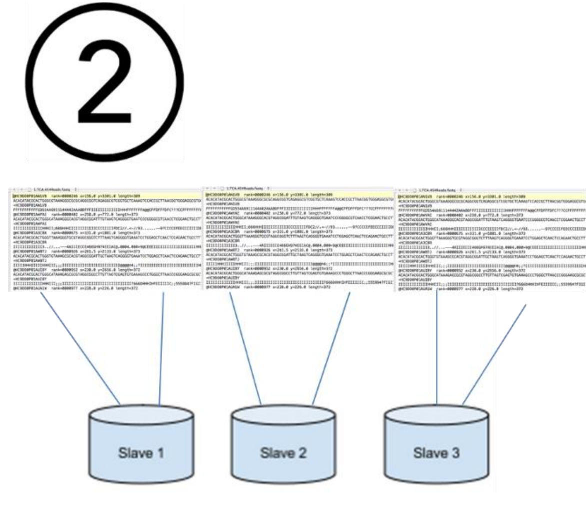
Per altra banda les conseqüències d'aquesta estratègia és que estem seqüenciant la lectura de manera que només un node està llegint i que fins que no s'omple el *buffer* per primera vegada els altres nodes no estan fent res. Com mostra la il·lustració també hi ha un conflicte amb la comunicació degut al alt nombre de dades que es transmeten i que si no es fa tenint en consideració com estan connectats els nodes podem arribar a saturar la xarxa provocant que els nodes estiguin més estona esperant dades que processant-les.

Estratègia 2

En la segona estratègia desapareix el concepte de *master* i passem a tenir un mapa en que tots els nodes fan la mateixa feina (veure il·lustració següent), que és la de llegir els arxius d'entrada. Per una banda ens estalviem haver d'enviar la informació a cada node, a més tots els nodes són aprofitats i treballen en paral·lel sense haver d'esperar els altres, el que ens aporta en principi, un bon nivell de paral·lelització.

El problema el trobem a l'accedir als arxius, si els arxius d'entrada estiguessin replicats en els discs dels diferents nodes seria perfecte, el problema és que és una idea utòpica ja que no és

viable de tenir uns arxius que s'utilitzaran una vegada (almenys en l'aplicació que estem construint) replicats en molts nodes i per tant hem de ser realistes. Degut a aquest entrebanc, tindríem que tots els nodes estaran competint per a llegir un mateix arxiu i que per tant estaríem seqüenciant la lectura i llegiríem els arxius tantes vegades com nodes utilitzem, a més caldria aplicar els mateixos filtres a cada node. De manera que al final no tindríem un nivell tant alt de paral·lelització com esperàvem. Hauríem serialitzat tant la lectura dels fitxers com el filtratge de qualitat, o sigui que a sobre és llegirien tantes vegades com nodes tenim i que en tots s'hi haurien d'aplicar els mateixos filtres.

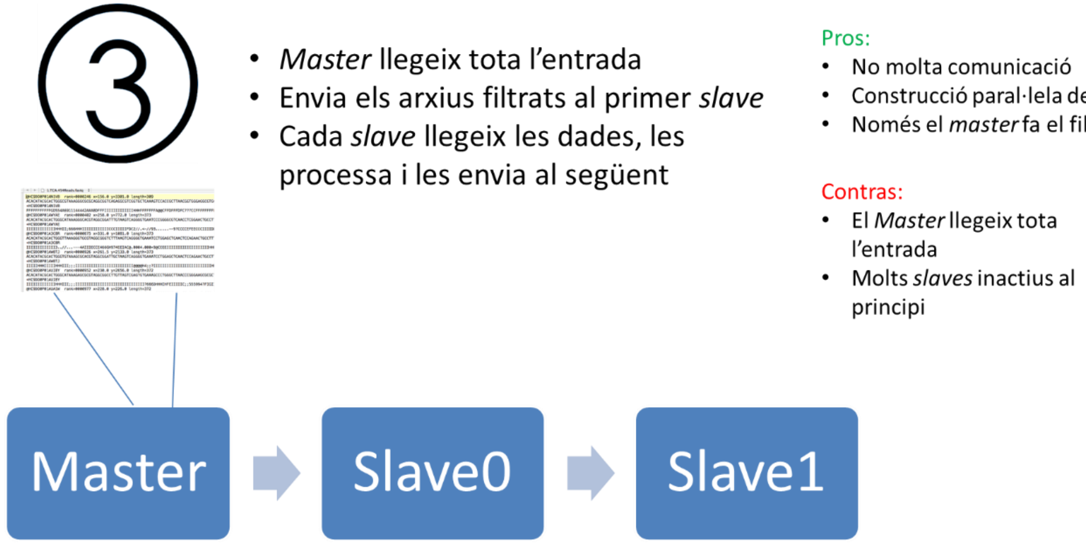


Il·lustració 8: Estratègia Inicial 2

Estratègia 3

En aquesta tercera estratègia torna a aparèixer el concepte de *master* (veure il·lustració següent), podríem dir que és molt semblant a la primera estratègia pel que fa a la lectura d'arxius però la diferència la trobem en la part de la comunicació. Aquí alliberem càrrega al *master* que només ha d'enviar els *reads* a un dels altres nodes, anomenats *slaves*. Com a punts forts doncs tenim que només llegim un cop l'entrada i a la part de la comunicació ja només enviem la informació necessària, havent filtrat la resta al *master*, i que la comunicació del *master* no es veu tant saturada com en la primera estratègia.

En contraposició ens trobem que si tenim n *slaves*, si definim el processat dels *reads* i enviament al següent node com una iteració, llavors l'*slave* n no començarà a treballar fins a la iteració n, això provoca que inicialment molts nodes estiguin inactius i que perdem eficiència en termes de paral·lelització, sobretot si utilitzem un gran nombre de nodes. Una correcció que podria millorar el problema és prioritzar en primer terme el transport de *reads* abans que el processat en cada node.



Il·lustració 9: Estratègia inicial 3

Un cop vistes les tres estratègies vàrem descartar la segona estratègia ja que la disposició que requeria dels arxius no era la que s'utilitzava al Mare Nostrum III, llavors ens vam quedar entre les dues que tenien *master*. Teníem la intuïció que la lectura seria més ràpida que el processat dels *reads*, és per això que ens vam decantar per sobrecarregar al *master* amb més feina i alliberar càrrega de treball als *slaves* i per tant vam apostar per la primera estratègia.

MPI

Pel que fa a MPI, durant l'execució es crea el que s'anomena un *MPI_Datatype* que no és més que la creació del paquet (com a estructura) que volem enviar de tal manera que es pugui enviar amb MPI. Per a això construíem el paquet molt simplement, es tracta de x paquets (on "x" és la mida del *buffer* que utilitzem d'enviament) que estant formats del *read* de mida fixa i de l'identificador. D'aquesta manera s'aprofita al màxim l'enviament ja que l'estructura que creem conté els camps justos que necessitem.

Com hem comentat abans l'enviament es fa de manera bloquejant, entre les dues opcions principals de *MPI: MPI_Send i MPI_SSend*, vam escollir la segona. La diferència entre les dues rau que en la segona no es fa realment l'enviament fins que no rebem el senyal que el node objectiu ha fet primer el MPI_Recv (rebuda bloquejant) corresponent, i per tant ens assegurem que es rebrà correctament. Amb *MPI_Send* podríen haver-hi complicacions amb tantes comunicació degut als *buffers* interns que utilitza *MPI.*

Anteriorment també enviàvem una altra senyal que era el nombre de parelles que contenia l'enviament, però vam descobrir que utilitzant l'estructura *MPI_Status* podíem obtenir aquest valor directament. Per a fer-ho cal utilitzar la comanda *MPI_GET_COUNT*, que ens indica el nombre d'elements enviats.

Un dels paràmetres de MPI_Recv és el de màxim número de paquets de tipus *datatype* que esperem rebre, en aquest camp doncs només cal posar-hi la mida màxima del buffer que serà el nombre màxim que rebrem, ja que tan el buffer del *master* i el dels *slaves* tenen la mateixa mida.

Una possible millora del programa seria la utilització de *MPI_Bcast*, que en principi està pensat per a reduir el temps que es tarda en comunicar un node amb la resta, com és el nostre cas.

Creació d'Arbres

Degut a termes biològics[15] els punts que busquem es troben a partir dels 30 (més o menys) nivells de profunditat el que comporta que si trobem una manera d'identificar els 30 nivells que tenim per sobre podem estalviar-nos d'inserir-los en l'arbre. La idea que se'ns va acudir és la de convertir els 30 primers nivells, que correspondrien als 30 primers nucleòtids, a un número, de manera que per qualsevol seqüència es pogués traduir a un nombre i donat aquest nombre també fóssim capaços d'obtenir la seqüència. Aquests 30 nucleòtids els vam anomenar "prefix", i el fet de guardar el prefix en format d'un número[16] ens permetia reduir la memòria utilitzada. La idea doncs de no inserir els primers 30 nivells ens comporta, que de cop tenim molts "subarbres" i que són identificables gràcies a aquest nombre.

Llavors un cop el grup de *reads* ha arribat a un *slave* aquest busca en tots els sufixes dins cada *read* i en el cas que algun d'ells estigui en el *rang* que ha de processar els insereix dins de l'arbre corresponent, per fer això analitzem per a cada sufix, el seu prefix (els x primers caràcters del sufix, per defecte 30, però es configurable amb el paràmetre --prefix_size) i el converteix amb un nombre. La fórmula és la següent:

$$\sum_{i=0}^{i=mida\,prefix} x^{4*i}$$

On x és:

- 0 si A
- 1 si C
- 2 si G
- 3 si T

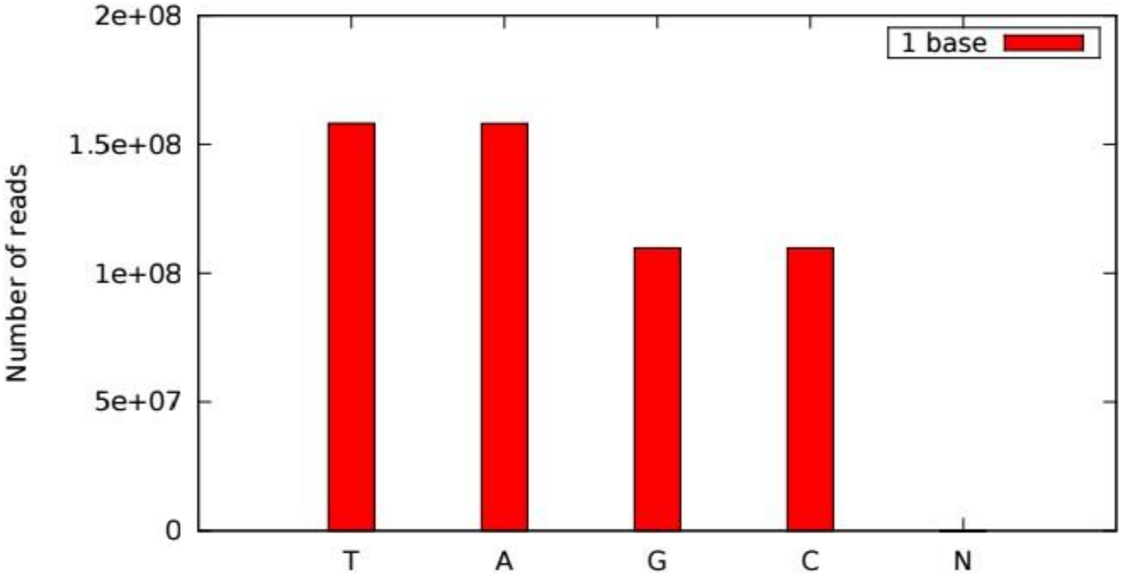
Inicialment vam fer una versió que dividia els prefixes de manera consecutiva: el primer *slave* és queda els j primers prefixes, el segon els j següents. On j és igual a:

$$j = \frac{número\ de\ prefixes}{número\ de\ slaves}$$

Però de seguida ens vam adonar que hi havia un desequilibri pel que fa a la càrrega dels prefixes, i això és degut a que en el nostre ADN apareixen més sovint alguns nucleòtids que d'altres, en concret A i T apareixen amb més freqüència, com es pot veure en la següent il·lustració, que és la distribució dels nucleòtids en el cromosoma vint-i-dos que nosaltres utilitzem:

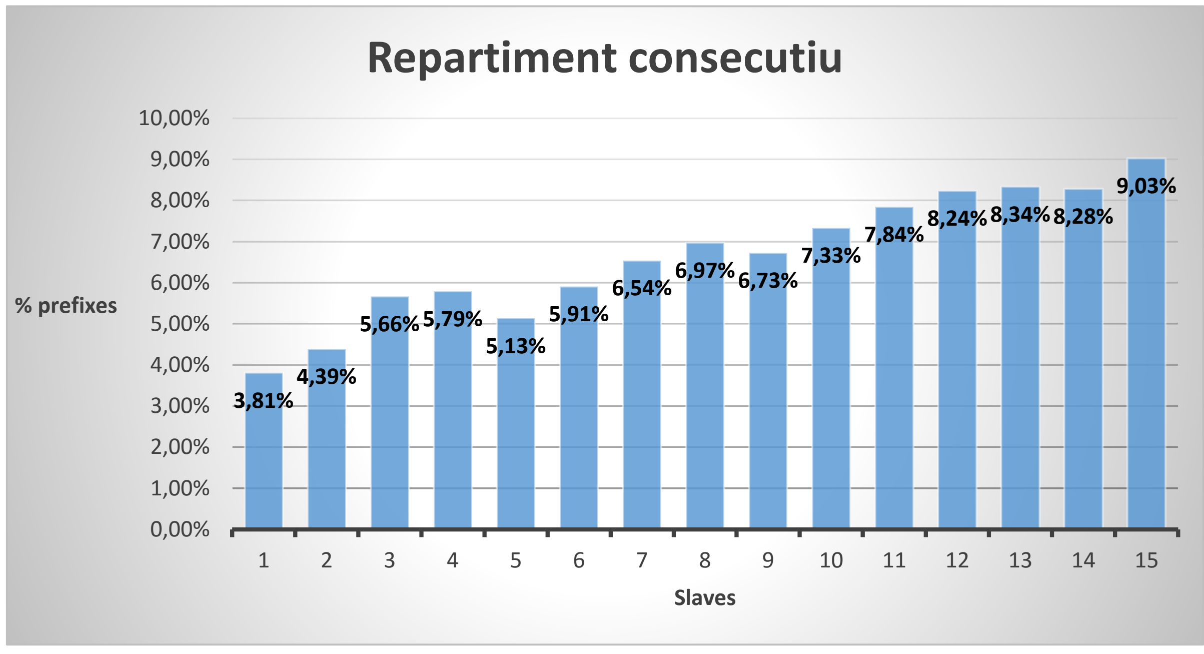
[15] Bàsicament a partir d'entre 25 i 35 nivells es considera que identifiquem un punt dins el genoma (unicitat), ja que si agaféssim per exemple un valor menor, com pot ser 4, amb 4 nucleòtids no seríem capaços de definir la zona on estem ja que el nombre de vegades que 4 nucleòtids en concret (per exemple ATGA) apareixen dins el genoma és molt elevat (poca unicitat).

[16] En concret un nombre de 64 bits, anomenat *"unsigned long long"* en C++, molt inferior a crear 30 nivells dins l'arbre.



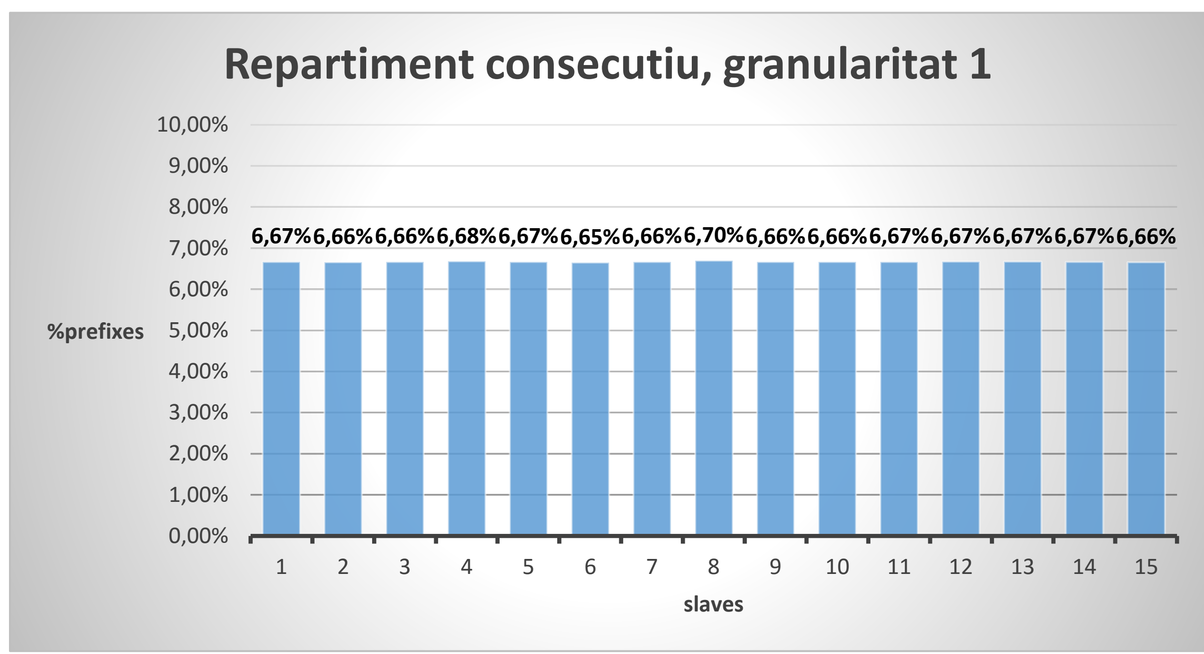
Il·lustració 10: Repartiment de nucleòtids en les dades d'entrada del cromosoma 22

Vegeu il·lustració següent que reflecteix el repartiment dels sufixes pel cromosoma vint-i-dos, executat en 16 nodes (15 *slaves* i 1 *master*):



Il·lustració 11: Distribució consecutiva

Com es pot veure la càrrega de treball no està ben repartida ja que tenim que el primer *slave* té prop del 4% dels *reads* mentre que el 15 en té un 9%. Per aquest motiu vam voler canviar la manera en què distribuíem els prefixes ja que això té una implicació directe amb l'ús de memòria que es fa en el node i amb la computació que aquest fa. La solució que se'ns va acudir va ser la de distribuir consecutivament però amb granularitat 1, això vol dir anar repartint un per un els prefixes: el primer pel primer *slave*, el segon pel segon *slave* ... i quan arribem al final tornem a començar. De manera que el què fem és fer el mòdul del prefix entre el nombre de *slaves*. El resultat amb les mateixes condicions que l'anterior va ser el següent:



Il·lustració 12: Distribució consecutiva, granularitat 1

Com es pot veure la distribució va ser un èxit total permeten així distribuir la càrrega de treball entre els nodes de manera simple i eficaç amb diferències màximes de 0'05%. Tot i que les proves només es van fer amb un cromosoma, en la posteritat s'ha comprovat que es continua mantenint aquesta distribució correctament, si s'utilitza un nombre no potència de 2 com a número de *slaves*.

Per a distribuir els arbres dins els *slaves* utilitzem un diccionari (o taula de hash). Per a designar aquest estructura vam pensar primer, en com serien els accessos als elements d'aquesta estructura. La resposta és fàcil són accessos aleatoris, això s'explica veient que mentre construïm l'arbre, l'ordre en què van arribant els *reads* és tal i com estant al fitxer d'entrada i per tant no es garanteix que vinguin de forma ordenada (s'entén ordenada com que els prefixes siguin consecutius), i a més per a cada *read* busquem tots els sufixes i els prefixes d'aquests tampoc estan ordenats, per tant com el seu nom indica farem accessos aleatoris al diccionari.

L'altre punt que buscàvem a l'estructura era que no utilitzés molta memòria, que sempre és un dels nostres objectius, i per tant destinar el màxim de memòria en els arbres. En definitiva necessitàvem una estructura que en termes de memòria no fos molt complexa, no volíem sobrecarregar la memòria amb l'estructura externa i per altra banda que potenciés els accessos

aleatoris en termes de velocitat d'accés i d'inserció. Amb aquest últim paràmetre com a principal punt a favor ens vam decidir en utilitzar el diccionari o taula de *hash*[17].

Per a definir l'estructura final vam escollir que la clau, fos el nombre que utilitzàvem per definir el prefix, ja que és una propietat única per a cada arbre i que necessitàvem guardar sí o sí. Per al camp valor que és el camp que realment té la informació que busquem vam posar-hi, no l'estructura de l'arbre directament, sinó simplement un punter a aquesta estructura ja que així evitem sobrecarregar l'estructura.

Durant l'elecció de l'estructura que utilitzaríem primer ens vam declinar cap a la taula de hash per defecte de *gcc* que s'anomena *unordered_map* però ens vam adonar que utilitzava bastanta memòria i buscant alternatives ens vam trobar amb un anàlisis [20] que bàsicament era un *benchmark* de taules de *hash* en què es comparaven diferents aspectes de l'estructura, com per exemple temps de lectura en accessos seqüencials, temps d'insercions, memòria utilitzada... Així que ens vam centrar en les gràfiques de temps d'inserció aleatòria i memòria utilitzada i en les dues teníem una vencedora clara diferent: en termes de insercions aleatòria: *dense_hash_map* (de *Google*) i per l'altra banda en memòria utilitzada: *sparse_hash_map* (també de *Google*). Com hem fet durant tot el projecte escollirem basant-nos primer amb la memòria utilitzada i és per això que finalment vam escollir la *sparse_hash_map.* Tot i que en l'anàlisi s'explica que és dues o tres vegades més lenta que l'altra alternativa, també es comenta que és 5 o 6 vegades menor en l'ús de memòria.

Així que resumint el què fa un *slave* al rebre un conjunt de *reads* és: mirar per a cada *read* tots els sufixes i si un d'ells l'hem d'incloure mirem si existeix l'arbre amb el mateix prefix dins l'estructura de hash, si no existeix el creem, i inserim el sufix dins l'arbre.

Obtenció de resultats i validació de les mutacions dins l'arbre

Un cop vam arribar a aquest punt vam adonar-nos que ens faltava una manera de validar l'Arbre (el conjunt de tots els arbres dels *slaves*), i comprovar que contenia totes les dades que havíem inserit. Com que estàvem treballant amb el cromosoma vint d'un «pacient irreal» ja que el material genètic amb el que treballàvem s'havia creat artificialment, sabíem en tot moment on hi havien les mutacions i de quin tipus es tractaven, de manera que ens va ajudar molt en la validació de l'aplicació.

Així que gràcies a que també teníem els arxius *BAM* del cromosoma, que ens permetien veure quins *reads* estaven involucrats en cada mutació, se'ns va acudir de fer l'estratègia que nosaltres vam anomenar *Read-Oriented.*

Read-Oriented

L'estratègia *Read-Oriented* es basa en la idea com hem comentat de que estem provant l'aplicació amb un joc de proves, el genoma "*In silico*", del qual ja coneixem les solucions i per tant ens permet validar el resultat fàcilment, l'objectiu doncs d'aquesta estratègia és només la de validar la creació de l'arbre i extreure dades del *coverage* en uns determinats punts.

---

[17] Una taula de hash és una estructura que associa dos conceptes: claus i valors. Per a fer-ho el què fa és: per a cada clau li aplica una "funció de hash", que no és més que una transformació en un número únic que identifica una posició, i per tant amb aquest número accedim el valor que li correspon a la clau.

El procés consisteix en buscar determinats sufixes que intervenen en una mutació i en concret aquells que no tenen la mutació en el seu prefix. Un cop trobats els punts dins l'arbre ens centrem en veure quin nombre de sufixes conservem, així que podrem comprovar si encara mantenim el mateix nombre de *coverage* i el més important si els *reads* associats a als punts són els correctes i per tant estan correctament inserits dins l'arbre, fet que vol dir que l'hem generat correctament.

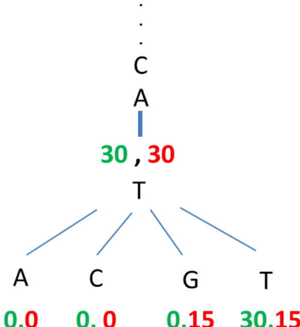
Per a realitzar el procés necessitem crear uns arxius d'entrada, nosaltres hem decidit dedicar un arxiu per a cada mutació i per tant tindrem tants arxius com punts de mutació. A l'interior dels arxius tindrem els *reads* tumorals associats a la mutació, i per cada sufix la posició de la mutació en el seu interior. L'encarregat de llegir els arxius serà com és natural el *master*, llavors per cada arxiu el *master* el llegirà, i enviarà als *slaves*: el sufix a buscar i el punt de mutació.

Els *slaves* per a cada *read* que rebin hauran de buscar tots els sufixes fins que el prefix arribi al punt de mutació, això es tradueix en que buscarem en tot l'arbre els punts en que tant els *reads* normals com els tumorals coincideixen i es divergeixen ja que és el punt exacte de la mutació. Imaginem que tenim la següent seqüència i que el punt vermell és el punt on es produeix la mutació [veure il·lustració següent]:

Read normal:	*Read* tumoral:
AGCAACGACGACGCATTCCAT	AGCAACGACGACGCATGCCAT

Il·lustració 13: Read normal i tumoral en un punt de mutació tipus Point Mutation

Llavors en l'arbre buscarem tots els primers sufixes, en què com a molt el prefix arribi a la "T" abans de la mutació, no té sentit fer-ne més ja que com que a partir d'aquest la mutació estarà dins el prefix farà que els sufixes del tumoral i el normal no caiguin a la mateixa branca, per exemple amb un prefix de 5: el normal seria GCATT mentre que el tumoral GCATG, prefixes diferents, branques diferents i nosaltres el què busquem és el punt just on divergeixen normal i tumoral. Suposant un cas idoni el resultat esperat seria:



Il·lustració 14: Cas idoni d'un punt de mutació

Com es pot veure suposant un *coverage* de 30, significaria que no hem perdut cap *read*, per suposat aquest és un cas ideal i és complicat que aparegui a la realitat. Per tal de visualitzar el

resultat de la cerca el què és, primer com que hem inserit tots els sufixes que anem a buscar ja que els *reads* que busquem provenen de la mateixa entrada, sabem que en tots tindrem algun resultat i que per tant el *master* espera rebre resposta de tots els sufixes. Per això el què fem és que el *master* fa un MPI_Recv per a cada sufix i per a identificar-los utilitzem el MPI_Tag ja que ens és independent quin *slave* ens ho enviï. La forma en que rebem la resposta és: primer rebem 10 comptadors que identifiquen els comptadors del node "pare", en el nostre cas la T (30,30) i els altres 8 corresponents a cada "fill". Seguidament rebem el nombre i el valor dels *id*, que identifiquen tots els *reads* involucrats en aquell punt.

La forma en que presentem els resultats és la següent:

A la part de dalt de l'arxiu hi posem la mateixa capçalera que a l'arxiu d'entrada, primer el punt de mutació segons el genoma de referència, després els *coverage* originals (obtinguts del *BAM*) després el sufix original que hem buscat i la posició de la mutació dins seu.

```
1788514
Number of tumor reads from bam  26 | with A 26 | with C 0 | with G 0 | with T 0
Number of normal reads from bam  18 | with A 18 | with C 0 | with G 0 | with T 0
GCCTCTCACAACGTCATGTACTTTTGAGTAAATGGGGTTACTTTAAGGTTAAGTGGAAAAAGAAAAGAGAATGAGAAAAA
78
```

Il·lustració 15: Capçalera de l'arxiu de sortida

Seguidament ja venen tots els sufixes ordenats de forma descendent en funció de la seva llargària. Per a cada sufix tindrem en aquest ordre: la mida del sufix, el sufix buscat, els 10 comptadors que corresponen com hem dit als comptadors del pare i dels fills, i els ids involucrats, on per a cada id tindrem l'arxiu i la línia dins l'arxiu on està el *read* que el conté:

```
38      CTTTAAGGTTAAGTGGAAAAAGAAAAGAGAATGAGAAA  18      18      18      11      0       0       0       0       0       7
x.out2.fq:6859173:49169591:/gpfs/scratch/bsc05/bsc05025/smufin/test_repo/inputs2/insilico/fastq/tumor/allele_1/allele_1_chr20.
ele_2/allel_2_chr20.30x.out1.fq:26486389:66287926:/gpfs/scratch/bsc05/bsc05025/smufin/test_repo/inputs2/insilico/fastq/tumor/a
nsilico/fastq/normal/allele_2/allel_2_chr20.30x.out1.fq:30849821:24559198:/gpfs/scratch/bsc05/bsc05025/smufin/test_repo/inputs
/test_repo/inputs2/insilico/fastq/tumor/allele_1/allele_1_chr20.fa.30x.out2.fq:39803849:53748840:/gpfs/scratch/bsc05/bsc05025/
ratch/bsc05/bsc05025/smufin/test_repo/inputs2/insilico/fastq/normal/allele_1/allel_1_chr20.30x.out1.fq:32693985:31475649:/gpfs
621:28438084:/gpfs/scratch/bsc05/bsc05025/smufin/test_repo/inputs2/insilico/fastq/normal/allele_2/allel_2_chr20.30x.out1.fq:12
r20.30x.out2.fq:13630973:56161504:/gpfs/scratch/bsc05/bsc05025/smufin/test_repo/inputs2/insilico/fastq/tumor/allele_1/allele_1
```

Il·lustració 16: Format d'exemple d'arxiu de sortida

Tree-Oriented

Com hem vist, l'estratègia del *Read-Oriented* només ens serveix pel cas del genoma "*In silico*", ja que sabem els sufixes exactes que hem de buscar per a trobar els punts de mutació. A la vida real doncs aquesta estratègia no és viable, i per això n'hem hagut d'idear una altra. La idea que hi ha darrera és molt senzilla, això si, que sigui senzilla no vol dir que sigui fàcil d'implementar ja que com veurem s'han de buscar uns filtres que permetin trobar tots els punts de mutació però intentant sempre de reduir el màxim possible els falsos positius.

La idea doncs, no és més que fer una cerca en tot l'arbre i aplicar uns determinats filtres en cada punt per determinar si aquest és un punt de mutació o no, per a fer-ho doncs ens fixarem principalment amb els comptadors del node en qüestió i els dels seus fills.

Pel que fa a la obtenció de resultats, tenim un problema i és que hem de trobar uns filtres adequats que ens treguin els falsos positius però que no filtri cap dels punts correctes. Per

aquesta idea, també ens hem servit que tenim els resultats del cromosoma amb el que treballem i per tant és intentar idear uns filtres partint dels punts trobats abans amb l'estratègia de *Read-Oriented* amb la gràcia que ara hem utilitzat aquesta mateixa estratègia en *reads* aleatoris, que sabem que no contenen cap mutació. D'aquesta manera podem provar diferents filtres mirant en tot moment el nombre de mutacions que aquests filtren i intentar ajustar-los al màxim per a filtrar els punts aleatoris, que serien falsos positius.

Així doncs ens guiem sobretot amb la hipòtesis que el genoma *In silico* pot simular un genoma natural i per tant que els passos que anem seguint i corroborant amb aquest són aplicables a la realitat. Val a dir però que hem de tenir en compte que no ho estem provant amb l'arbre degut a la facilitat que ens aporta treballar només amb aquests punts, però tenim en compte que els falsos positius poden ser més nombrosos en l'arbre ja que només n'hem agafat un subconjunt.

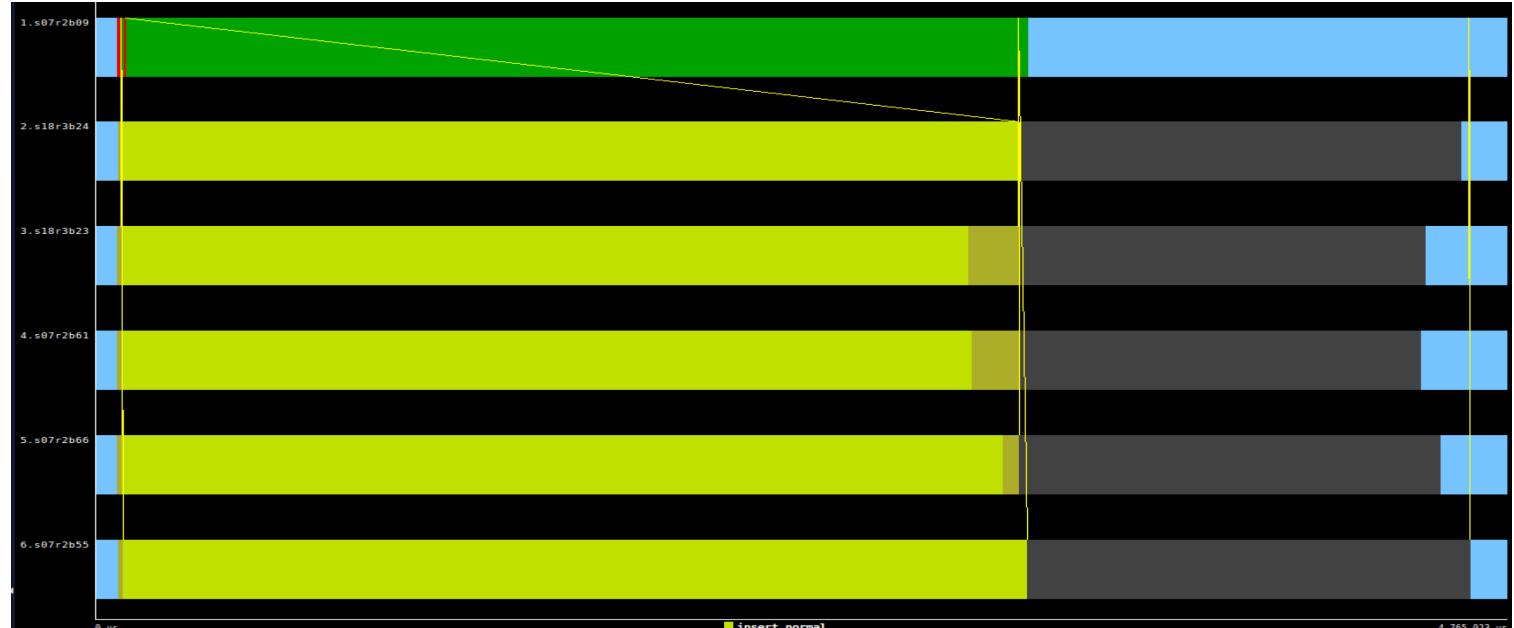
De moment encara estem treballant en buscar uns bons filtres, en aquesta part hem tingut l'ajuda d'alguns dels biòlegs i biotecnòlegs del grup on ens trobem ja que aquí podríem dir que es tornen a encreuar la feina d'informàtic i d'altres disciplines.

4.3 Anàlisi

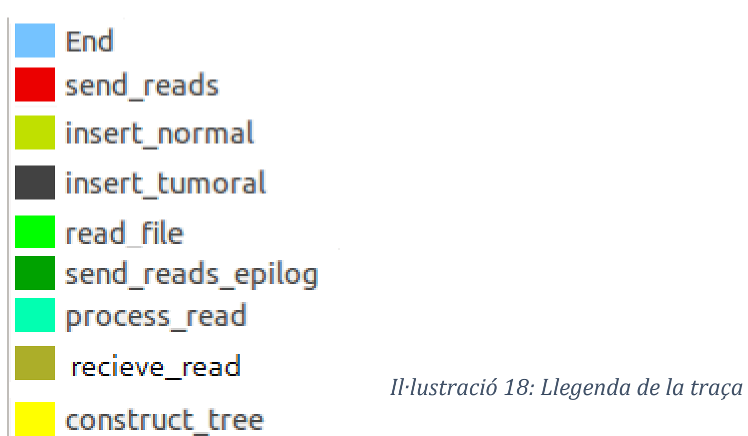
Un cop acabada la primera versió vam instrumentar-la per a poder avaluar-ne el rendiment i veure els seus punts , que serien els que prioritzaríem alhora de buscar optimitzacions. Degut al gran detall que volíem arribar amb l'anàlisi de l'execució per tal de no crear arxius per analitzar immensos hem decidit ajustar els paràmetres del programa per que s'adeqüin millor. En concret hem reduït el nombre de nodes utilitzats i els arxius d'entrada han estat substituïts per uns de més petits. Les traces han estat generades utilitzant l'aplicació *Extrae* i visualitzades gràcies a l'eina *Paraver,* les característiques de l'execució es mostren a continuació:

- Nombre de nodes: 6 (5 *slaves* i 1 *master*)
- Total dades d'entrada més reduïda de 15,6 GB (cromosoma 20) a 1790 KB
- *Readblock* : 50 MB
- Mida *buffer* d'enviament: 6,48 MB (60 *reads*)
- A més només entrem fastq1 tant de normal com tumoral i per tant ni fastq2 de normal ni de tumoral

A continuació veurem la traça general de l'aplicació, tot il·lustrada amb les funcions i les comunicacions entre els diferents nodes:



Il·lustració 17: Traça General de l'execució



Il·lustració 18: Llegenda de la traça

Cada color identifica una funció dins els codi com podem veure a la llegenda, a continuació explicarem què es fa en cada funció per a tal de poder comentar la gràfica:

- *End*: Té varis significats, la primera part ens indica el pas de paràmetres inicials, mentre que la part del final indica primer que els *slaves* esperen la senyal de comunicació que indica que s'han acabat els arxius d'entrada (les línies grogues primes representen la comunicació), i un cop rebuda aquesta senyal representa que els nodes (tant *master* com *slaves*) eliminen les estructures i s'esperen a acabar tots junts.
- *Send_reads(master)*: La funció send_reads és l'encarregada de copiar els *reads* llegits al buffer d'enviament, i si aquest s'omple enviar-los als *slaves*
- *Insert_normal(slaves)*: Funció encarregada d'inserir un sufix dins l'arbre que li correspon.
- *Insert_tumoral(slaves)*: Ídem que l'anterior però pel cas dels sufixes tumorals.
- *Read_file(master)*: Funció encarregada com el seu nom indica de llegir un arxiu d'entrada, en concret de llegir blocs d'aquests de mida fixada amb el paràmetre

readblock i processar-los, tot cridant al send_reads per a cada *read* trobat i que passa el filtre de qualitat.

- *Send_reads_epilog(master)*: Aquesta funció és l'encarregada d'enviar l'últim paquet de *reads* tant si el *buffer* d'enviament està ple com si no.
- *Process_read (slaves):* Encarregada de processar els reads
- *Recieve_read(slaves):*Encarregada de rebre els reads i es dóna per finalitzada al acabar un arxiu
- *Construct_tree(master):*Funció que crida a les funcions de lectura dels arxius d'entrada

Cal esmentar que quan l'aplicació comença ja hi ha una primera comunicació (línia groga tant fina que no es veu, però existeix) que és el pas de paràmetres inicials del *master* (1a fila), als *slaves* aquesta comunicació difereix de les altres en que es fa utilitzant la comanda de MPI : MPI_Bcast, que és més eficient que enviar un MPI_Ssend a cada node. És per això que la durada és tant petita que n'hi es mostra en la 1a traça. Tot i això sabem que s'han produït les crides gràcies a la següent taula que ens mostra el nombre de crides a funcions *MPI:*

	MPI_Recv	MPI_Bcast	MPI_Barrier	MPI_Comm_rank	MPI_Comm_size	MPI_Init	MPI_Finalize	MPI_Ssend
1.s07r2b09	-	9	2	7	7	1	1	15
2.s18r3b24	3	9	2	7.473	7.473	1	1	-
3.s18r3b23	3	9	2	7.473	7.473	1	1	-
4.s07r2b61	3	9	2	7.473	7.473	1	1	-
5.s07r2b66	3	9	2	7.473	7.473	1	1	-
6.s07r2b55	3	9	2	7.473	7.473	1	1	-
Total	15	54	12	37.372	37.372	6	6	15

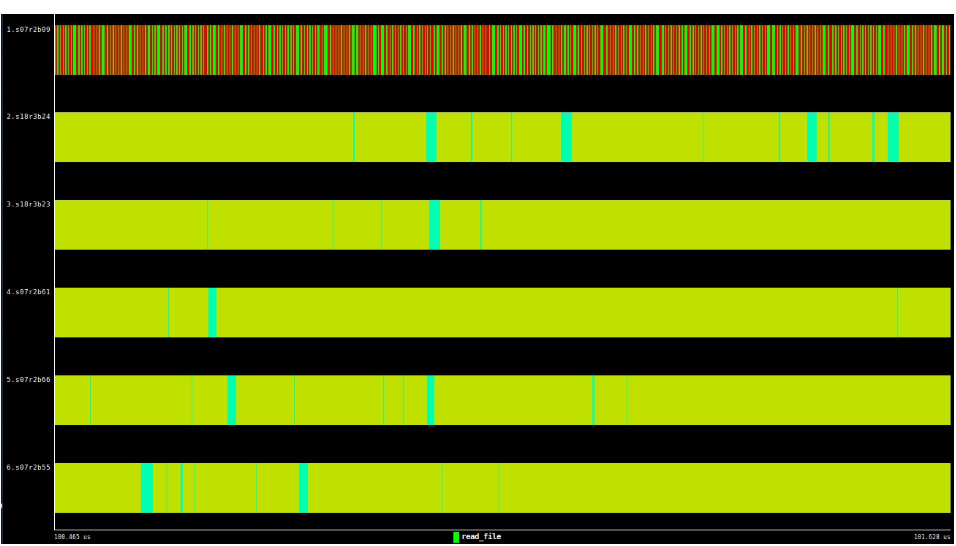
Taula 12: Nombre de crides MPI (versió inicial)

Com veiem a la taula s'han produït els *Broadcasts* que esmentàvem, veiem també que el nombre de vegades que cridem a les funcions *MPI_Comm_rank* i *MPI_Comm_size* que s'utilitzen per identificar quin node és i quants nodes hi ha respectivament, és molt elevat, el motiu principal és perquè cada vegada que inserim un *read* necessitem saber quin node sóc i quants n'hi ha per escollir si inserir el sufix o no (perquè ja ho fa un altre). Veiem també a la següent taula que la funció MPI on s'hi destina més temps és la MPI_Ssend i és comprensible ja que és la que envia tots *reads* als altres nodes, i com hem comentat abans en el cas de que el node *worker* no estigui preparat per rebre, el *master* queda parat en aquell lloc i per tant compta també com a temps. També és interessant veure que el temps dels MPI_Recv és molt més elevat que els MPI_Bcast tot i que s'utilitzen menys vegades però cal matissar que els MPI_Recv reben una quantitat de dades més gran que no pas els MPI_Bcast. Comentar que els temps del *Barrier* (barrera) ens indiquen quins són els nodes amb més càrrega de treball tot veient el temps dedicat, sent el *master* el menor en càrrega i el *worker* el que més.

	MPI_Recv	MPI_Bcast	MPI_Barrier	MPI_Comm_rank	MPI_Comm_size	MPI_Init	MPI_Finalize	MPI_Ssend
1.s07r2b09	-	192,78 us	1.491.253,67 us	175,50 us	38,29 us	30,48 us	9,55 us	3.040.322,98 us
2.s18r3b24	15.108,89 us	2.260,28 us	31.574,18 us	33.784,99 us	30.781,03 us	311,05 us	9,41 us	-
3.s18r3b23	190.210,00 us	185,98 us	150.439,78 us	34.930,74 us	31.073,96 us	314,15 us	10,24 us	-
4.s07r2b61	176.733,41 us	2.248,50 us	165.268,75 us	34.106,40 us	30.735,26 us	308,39 us	9,50 us	-
5.s07r2b66	73.978,15 us	2.259,72 us	99.587,09 us	33.394,60 us	30.338,39 us	307,44 us	9,41 us	-
6.s07r2b55	14.322,57 us	247,19 us	79,08 us	34.026,65 us	30.774,97 us	310,33 us	9,17 us	-
Total	470.353,01 us	7.394,45 us	1.938.202,55 us	170.418,88 us	153.741,90 us	1.581,84 us	57,28 us	3.040.322,98 us
Average	94.070,60 us	1.232,41 us	323.033,76 us	28.403,15 us	25.623,65 us	263,64 us	9,55 us	3.040.322,98 us
Maximum	190.210,00 us	2.260,28 us	1.491.253,67 us	34.930,74 us	31.073,96 us	314,15 us	10,24 us	3.040.322,98 us
Minimum	14.322,57 us	185,98 us	79,08 us	175,50 us	38,29 us	30,48 us	9,17 us	3.040.322,98 us
StDev	76.255,27 us	1.023,95 us	525.768,18 us	12.632,23 us	11.444,13 us	104,30 us	0,33 us	0 us
Avg/Max	0,49	0,55	0,22	0,81	0,82	0,84	0,93	1

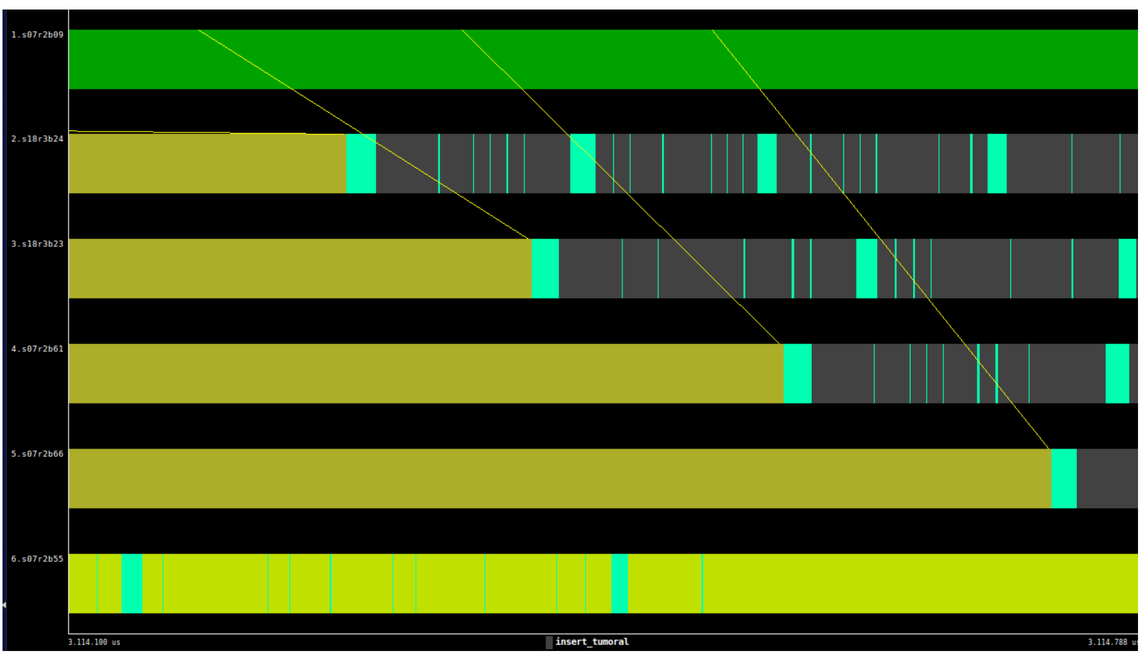
Taula 13: Taula amb els temps de les crides a MPI

Un cop el node *master* ha llegit tot l'arxiu, com que el buffer d'enviament no s'ha omplert cap cop encara no hem fet cap enviament, és llavors quan cridem a la funció de "send_reads_epilog" i aquesta envia tots els *reads* llegits en el primer arxiu cap als nodes que han estat esperant *reads*. En aquest punt els *slaves* insereixen els sufixes que els hi pertoca dels *reads* rebuts. La gran franja groga dels *slaves* indica que estan a la funció "insert_normal". Però al fer una mica de zoom, com veiem a la següent il·lustració ens mostra que realment dins la franja groga hi ha franges més estretes de color turquesa que corresponen a l'anàlisi del *read* ("process_read"). Això ens indica que en l'execució dels *slaves*, la major part del temps es dedica a la inserció dels sufixes, segurament degut als nombrosos accessos a memòria que cal fer per a poder inserir tots els nodes. En aquesta part també veiem que tot i que en la primera imatge veiem el *master* en una gran franja verda, en realitat es tracta d'un conjunt de funcions que en definitiva llegeixen el segon fitxer i a l'acabar, espera que tots els nodes estiguin esperant el següent enviament per a tornar-los-hi a enviar *reads*.



Il·lustració 19: Zoom inserció normal

Una cosa que pot cridar l'atenció en la primera traça, és que tot i ser els dos fitxers d'entrada de la mateixa mida, el temps d'inserció de la part normal és més extensa que la part tumoral, aquesta particularitat és degut a unes optimitzacions d'insercions que fan que en alguns casos no sigui necessari inserir tots els sufixes del *reads* tumorals. En la següent il·lustració veiem un zoom de la part de la inserció tumoral. Aquí veiem doncs que la presència de turqueses és un pèl més elevada que en el cas del normal, a més veiem perfectament que el *master* no envia els *reads* fins que el node esclau no ha acabat el procés anterior. La zona beix indica que els nodes esclaus estan esperant els nous *reads.*



Il·lustració 20: Zoom inserció tumoral

A continuació veurem unes taules estadístiques d'aquesta execució[18]:

[18] Dades extretes de l'aplicació Paraver

	Idle	Running
1.s07r2b09	97,80 %	2,20 %
2.s18r3b24	5,18 %	94,82 %
3.s18r3b23	11,32 %	88,68 %
4.s07r2b61	11,34 %	88,66 %
5.s07r2b66	7,80 %	92,20 %
6.s07r2b55	5,17 %	94,83 %
Total	138,62 %	461,38 %
Average	23,10 %	76,90 %
Maximum	97,80 %	94,83 %
Minimum	5,17 %	2,20 %
StDev	33,50 %	33,50 %
Avg/Max	0,24	0,81

Taula 15: Estat dels processadors durant l'execució

Com veiem a la taula pel que fa als nodes esclaus tenim un rendiment bastant bo, ja que el *worker* que més estona està "parat" (*idle*), hi està un 11,34% del temps, pel que fa al node *master* tenim el cas oposat ja que només està funcionant un 2,2% del temps. Si definim el màxim d'eficiència que podríem obtenir com que tots 6 nodes estiguessin el 100% en "*running*", i per tant el total ideal estigués al 600%, si extrapolem aquesta dada per treure una referència de l'eficiència tenim que la nostra aplicació té una eficiència de 461,38/600 que equival a la mitjana mostrada a la taula (76,90%). Una altra manera d'avaluar el resultat és veure percentatge de temps destinat a cada funció com veiem a la taula següent (*master* a la primera fila):

	read_file	send_reads	send_reads_epilog	recieve_reads	process_read	insert_normal	insert_tumoral
1.s07r2b09	0,69 %	0,34 %	98,97 %	-	-	-	-
2.s18r3b24	-	-	-	0,38 %	3,96 %	64,67 %	30,99 %
3.s18r3b23	-	-	-	4,36 %	4,10 %	62,49 %	29,05 %
4.s07r2b61	-	-	-	4,07 %	4,10 %	62,98 %	28,85 %
5.s07r2b66	-	-	-	1,70 %	4,00 %	64,38 %	29,91 %
6.s07r2b55	-	-	-	0,36 %	3,94 %	64,91 %	30,79 %
Total	0,69 %	0,34 %	98,97 %	10,87 %	20,10 %	319,44 %	149,59 %
Average	0,69 %	0,34 %	98,97 %	2,17 %	4,02 %	63,89 %	29,92 %
Maximum	0,69 %	0,34 %	98,97 %	4,36 %	4,10 %	64,91 %	30,99 %
Minimum	0,69 %	0,34 %	98,97 %	0,36 %	3,94 %	62,49 %	28,85 %
StDev	0 %	0 %	0 %	1,74 %	0,07 %	0,97 %	0,87 %
Avg/Max	1	1	1	0,50	0,98	0,98	0,97

Taula 16: Temps en % dedicat per funció

Com veiem en la taula, el *master* està molta estona en la funció d'enviament esperant que els nodes *slaves* hagin acabat d'inserir tots els sufixes dels *reads* passats anteriorment, per a poder-

los-hi enviar els nous. En canvi pel què fa els *slaves* veiem com efectivament estant molta estona inserint sufixes o processant *reads* i poca estona esperant els nous *reads*, que seria un dels components de l'estat "*Idle"* que és el cas de la funció "*recieve_reads*".

En conclusió veiem que l'execució té un bon rendiment pel que fa als node *slaves* ja que entre ells hi ha una eficiència del 91'8%, però pel que fa al *master*, aquest decaigui en un 2,2%, provocant així que l'eficiència total baixi fins al 76,9%. Tot i això estem satisfets amb la primera versió perquè preferim que tinguin millor rendiment els nodes *slaves* que el *master* ja que el nombre sempre es superior, en aquest cas de 5 a 1, i per tant a costa de desaprofitar un node en potenciem 5, així que com més grans siguin els fitxers d'entrada i per tant requerim de més nodes l'eficiència de l'aplicació tendirà a créixer ja que de *master* sempre en tindrem només 1. Ens apuntem però que l'eficiència del *master* és molt petita i per tant haurem de buscar algun tipus d'optimització per a potenciar-la, com per exemple fer que el *master* aprofiti aquell temps per crear S*uffix Tree* ajudant d'aquesta manera als *slaves* tot reduint-los-hi la càrrega de treball.

Tot i que veient les gràfiques el temps de comunicació no sembla molt rellevant també seria interessant de millorar aquest apartat amb l'ús de la comanda MPI_Bcast, que és una funció de MPI especialitzada en la distribució de dades d'un sol node a la resta, que en definitiva és el que estem fent, però que té un rendiment millor que el MPI_Ssend que utilitzem actualment.

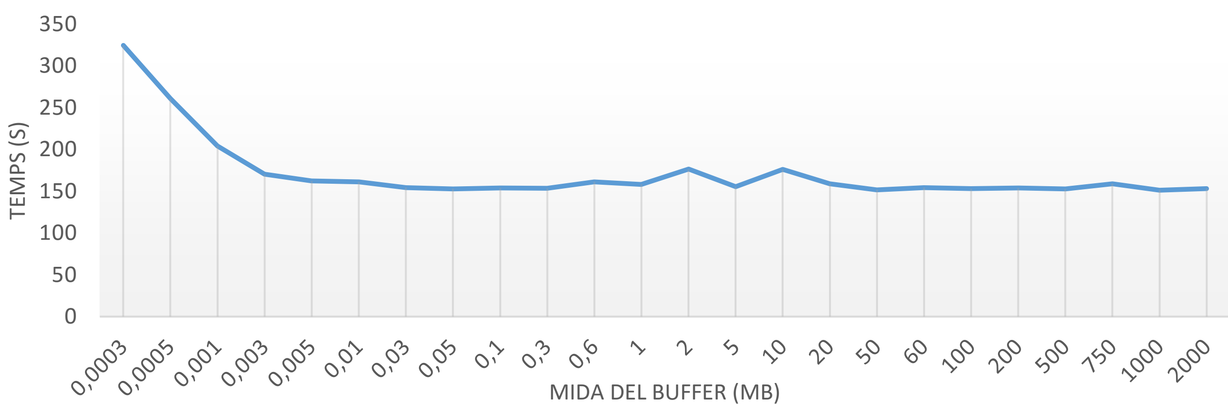
Un altre punt que ens interessava avaluar era quin impacte tenien la mida del *buffer* de lectura i d'enviament en l'execució.

Per una part el *buffer* de lectura és el responsable del nombre de lectures a disc que fa al *master* al llegir els arxius d'entrada. Un *buffer* de lectura s'utilitza per accelerar les lectures a disc mitjançant lectures més grans, d'aquesta manera aprofitem la sobrecàrrega que es produeix al llegir del disc per fer una lectura més gran, així en les lectures posteriors es llegirà directament del *buffer* que està emmagatzemat a memòria i que té un temps d'accés molt més reduït en comparació al disc.

D'aquesta manera la mida d'aquest *buffer* és clau ja que pot ésser fatal el temps de lectura només canviant aquest únic valor. Per exemple si escollim una mida massa petita, la sobrecàrrega del disc no serà tant aprofitada fins al punt que moure del disc al buffer i del buffer al programa sigui més lent que directament del disc al programa, per altra banda si escollim una mida molt gran estaríem reduint la memòria disponible pel sistema podent arribar a produir *swapping*[19]*,* el que fa endarrerir encara més el procés de lectura.

A continuació podem veure una gràfica de l'impacte que té la mida del *buffer* de lectura (*readblock*) en la nostra aplicació, en concret a tot el procés de lectura sense tenir en compte l'enviament de *reads*:

[19] L'espai de intercanvi és una zona del disc que s'utilitza per emmagatzemar zones de memòria de processos que no s'estan executant i que no necessiten aquestes dades en la memòria física o que no hi caben . Aquest espai s'acostuma a anomenar "swap" de l'anglès intercanviar.



Il·lustració 22: Temps de lectura dels fitxers del cromosoma 20 segons la mida del buffer de lectura

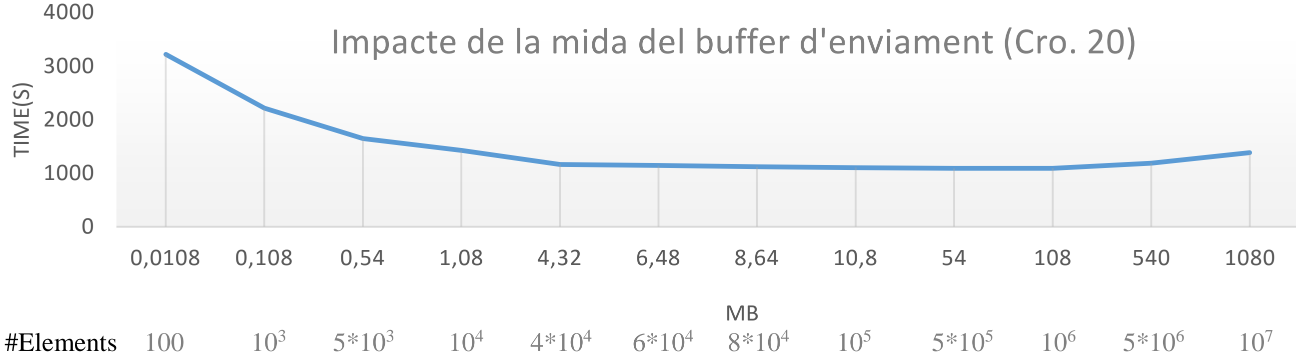
A l'inici es produeix el fenomen que explicàvem abans i veiem com tenim un temps superior a la resta. A l'augmentar la mida, el problema es soluciona i arribem a un punt en què la mida del *buffer* ja no produeix canvis representatius al temps de lectura. Malauradament no hem arribat al punt en que els temps tornin a pujar degut a una mida excessiva del *buffer*, ja que hem cregut que amb una mida de 2GB era suficient. Així que en definitiva podem concloure que mentre utilitzem una mida superior a 0,005MB (5KB) serà suficient per evitar una lectura ineficient.

L'altre buffer que pot modificar el comportament de la nostra aplicació és el *buffer* d'enviament, que utilitza el *master* per, com el seu nom indica, enviar els *reads* juntament amb els *ids* a la resta de nodes.

Les conseqüències que pot provocar la mida d'aquest *buffer* en la nostra aplicació són diferents respecte si és més gran o més petit de la seva mida ideal però en ambdós casos és perjudicial per a l'eficient execució del programa. Recordem que un cop el *buffer* està ple, aquest indica al *master* que s'ha d'enviar tot el contingut que aquest conté, i per tant és determinant alhora de decidir la freqüència en que s'envien els *reads*.

Per una banda, si tenim una mida superior a la idònia, estarem provocant que la resta de nodes (*slaves*) inicialment estiguin molta estona inactius esperant el primer enviament de *reads* i més endavant es repeteix-hi aquest procés, o sigui que el node *master* continuï llegint mentre els nodes estan inactius esperant nous *reads*. Seguint en l'altra direcció, si aquest és massa petit podem trobar-nos que el node *master* acabi dedicant més temps a enviar els *reads* (que seran pocs en cada enviament) que en llegir, provocant doncs un augment en el temps de lectura dels arxius.

La següent gràfica ens il·lustra sobre el comportament que acabem de descriure, en concret del temps d'execució total de l'aplicació en el cromosoma 20 només variant la mida del *buffer*:



Il·lustració 23: Temps d'execució en funció de la mida del buffer d'enviament

Les mides del *buffer* han estat seleccionades en funció del nombre d'elements (*read* + *id*) que hi caben que és el segon eix que hem indicat a sota del de MB, que va des de 100 fins a 10^7 elements. En la gràfica s'il·lustra perfectament els dos efectes descrits prèviament veient que s'estabilitza a partir dels $5*10^3$ elements i que comença a augmentar de nou als $5*10^6$. Gràcies a aquest anàlisi ara ja sabem quins són els valors indicats pels nostres *buffers* de manera que sigui més eficient tant la lectura com l'enviament de dades.

4.4 Millores

4.4.1 *Master* amb *Suffix Tree (Master Tree)*

Com hem comentat anteriorment un cop vist les estadístiques i les traces de l'execució de la primera versió, sobretot la de l'eficiència, ens hem plantejat la idea que el node *master* també insereix-hi *reads*, reduint així la càrrega de treball dels *slaves* i de retruc augmentar l'eficiència del *master*, que té un valor bastant baix.

L'estratègia que hem seguit és que un cop al *master* envia els *reads* als *slaves*, aquest també insereix els mateixos *reads* que acaba d'enviar (evidentment inserirà els sufixes que li pertoquin). La taula següent, realitzada en el joc de proves reduït, utilitzat en el cas de la versió inicial, mostra els resultats obtinguts d'eficiència:

	Idle	Running
THREAD 1.1.1	21,28 %	78,72 %
THREAD 1.2.1	4,91 %	95,09 %
THREAD 1.3.1	30,81 %	69,19 %
THREAD 1.4.1	9,03 %	90,97 %
THREAD 1.5.1	27,56 %	72,44 %
THREAD 1.6.1	14,61 %	85,39 %
Total	108,20 %	491,80 %
Average	18,03 %	81,97 %
Maximum	30,81 %	95,09 %
Minimum	4,91 %	69,19 %
StDev	9,39 %	9,39 %
Avg/Max	0,59	0,86

Taula 17: Estat dels processadors durant l'execució amb Master Tree

Veiem que la taula ha canviat bastant, la gran millora la veiem, com era previsible en el node *master* que augmenta d'un 2,2% a un 78,7% l'estat de *running*, tot i que quasi tots els *slaves* empitjoren el resultat final és d'un 81,97% d'eficiència superant així el 76,9% de la versió inicial.

Un cop vista l'estadística vam decidir de provar aquesta nova millora amb el joc de proves gran (cromosoma 20) i malauradament vam veure com els resultats que aquesta obtenia eren inferiors en quant a temps ja que teníem un increment del 81%. És per això que tot i que en els jocs de prova més petits tingui millors resultats (un 5% més ràpid), nosaltres hem de tenir en consideració que l'aplicació s'utilitza per a grans volums de dades i el resultat per aquests és negatiu. En definitiva no hem inclòs aquesta optimització en l'aplicació final.

4.4.2 Partitions

Una altra estratègia que se'ns va acudir va ser l'anomenada *partitions*. Aquesta es basa en la idea que quan analitzem l'arbre per a buscar possibles punts de mutació, mirem un punt en concret de l'arbre i la resta de l'arbre no té cap implicació en aquell moment o sigui que és independent de la resta de l'arbre. Gràcies a aquesta característica, podem construir per exemple la meitat de l'arbre, analitzar-la i un cop analitzada, eliminar aquesta part i fer el mateix per l'altra part. O anar més endavant i dividir l'arbre en tantes parts com vulguem.

Aquesta és la part que nosaltres anomenem *partitions,* el nombre de particions que aplicarem a l'arbre i que és un paràmetre configurable. D'aquesta manera si per exemple posem un *partitions* de 5, estem dividint la memòria que necessitem en 5 (cas ideal) i per tant estem eliminant la limitació de memòria que poden tenir algunes màquines per a executar l'aplicació, val a dir però que implica un increment en el temps que tarda l'aplicació degut entre altres que s'han de llegir les entrades x vegades (5 en l'exemple) i que s'han d'eliminar les estructures creades cada x vegades també. Així podríem dir que aquesta optimització ens permet alternar

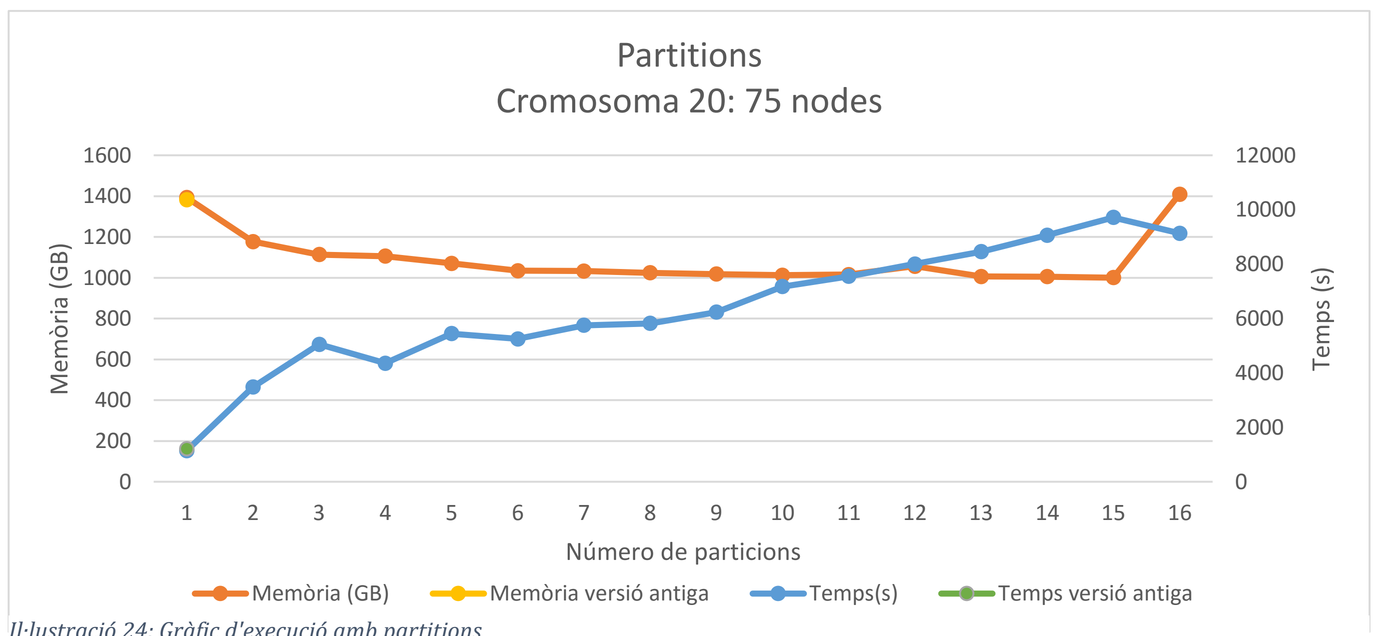
la memòria necessitada per l'aplicació segons el requeriment de les màquines on s'executa tot saben que una reducció d'aquesta implica un increment pel que fa al temps.

A més aquesta característica podem aplicar-la als estudis poblacionals, tot i l'impacte en el temps d'execució, la memòria ja no serà el problema principal a tenir en consideració, que fins ara era el coll d'ampolla principal que podíem trobar-hi.

Per a poder realitzar aquesta nova característica dividim del rang total de prefixes que existeixen entre el nombre de *partitions* que s'ha configurat, d'aquesta part n'anomenen un bloc de treball. En cada bloc de treball tenim un prefix mínim i un de màxim que identifiquen el conjunt de prefixes a tractar en aquella iteració, tenint en compte que l'última pot tenir un pèl més de càrrega de treball. Llavors quan els *slaves* reben un *read* i en revisen els prefixes el primer que comproven és si està dins els rang d'aquell bloc de treball.

En definitiva aquesta nova característica de l'aplicació farà que sigui més portable, al poder modificar el consum de memòria utilitzat (més *partitions* -> menys memòria requerida) permeten així d'executar l'aplicació en clústers que no tinguin tanta memòria disponible (podríem dir que treu la limitació de memòria), el que suposa un gran avanç pel que fa a la part dedicada als estudis poblacionals.

A continuació tenim un gràfic amb els resultats d'executar el cromosoma 20 del genoma "*In silico"* utilitzant 75 nodes. Com a guia hem posat també els valors de la versió anterior per veure que en cas d'utilitzar una sola partició no tenim cap pèrdua de rendiment.



Il·lustració 24: Gràfic d'execució amb partitions

Com podem veure el comportament és l'esperat, a mesura que augmentem les particions es redueix la memòria utilitzada i augmenta el temps d'execució. Tot i això ens esperàvem una reducció pel que fa a memòria més elevada. Encara que sembli que el punt òptim és entre 11 i 12 particions, cal recordar que cada una de les sèries de valors té la seva escala diferent al costat i si ens hi fixem veiem que un possible punt òptim seria el 4, ja que té una reducció en el temps comparat amb el seu anterior a part de la de memòria. Creiem que això és producte d'una distribució incorrecte pel que fa a càrrega de treball entre els diferents blocs de treball o particions. A més és important esmentar que l'arbre que utilitzem té un comportament peculiar

quan inserim sufixes, i és que sempre intentem tenir els sufixes comprimits i per tant a mesura que augmentem el nombre de sufixes inserits, no només augmentem la memòria degut a aquesta descompressió que cal fer sinó que també augmentem el temps d'inserció ja que cal anar descomprimint per tal de poder inserir. Així que a l'aplicar l'optimització de *partitions*, també estem reduint en certa mesura aquest comportament i per tant reduint el temps d'inserció en l'arbre gràcies a l'efecte que acabem de descriure.

Una altra comparativa que ens agradaria mostrar no només amb la versió inicial sinó també amb la primera optimització *Master Tree*, és la de l'eficiència, els resultats extrets per 2, 4, 8 i 16 són els següents:

	Idle	Running
1.s05r2b37	23,41 %	76,59 %
2.s09r2b10	14,55 %	85,45 %
3.s09r2b16	29,81 %	70,19 %
4.s13r1b77	4,70 %	95,30 %
5.s13r1b78	30,48 %	69,52 %
6.s14r2b16	13,17 %	86,83 %
Total	116,12 %	483,88 %
Average	19,35 %	80,65 %
Maximum	30,48 %	95,30 %
Minimum	4,70 %	69,52 %
StDev	9,36 %	9,36 %
Avg/Max	0,63	0,85

Taula 19: Estat dels processadors, partitions 2

	Idle	Running
1.s04r1b64	14,62 %	85,38 %
2.s09r2b77	11,01 %	88,99 %
3.s09r2b04	29,91 %	70,09 %
4.s12r2b45	7,14 %	92,86 %
5.s15r1b48	33,84 %	66,16 %
6.s16r2b45	16,73 %	83,27 %
Total	113,24 %	486,76 %
Average	18,87 %	81,13 %
Maximum	33,84 %	92,86 %
Minimum	7,14 %	66,16 %
StDev	9,73 %	9,73 %
Avg/Max	0,56	0,87

Taula 18: Estat dels processadors, partitions 4

	Idle	Running
1.s05r1b82	18,16 %	81,84 %
2.s05r2b49	15,06 %	84,94 %
3.s05r2b55	32,04 %	67,96 %
4.s05r2b36	5,92 %	94,08 %
5.s05r2b37	26,70 %	73,30 %
6.s06r1b64	15,04 %	84,96 %
Total	112,92 %	487,08 %
Average	18,82 %	81,18 %
Maximum	32,04 %	94,08 %
Minimum	5,92 %	67,96 %
StDev	8,48 %	8,48 %
Avg/Max	0,59	0,86

Taula 21: Estat dels processadors, partitions 8

	Idle	Running
1.s05r2b37	19,89 %	80,11 %
2.s09r2b16	6,32 %	93,68 %
3.s09r2b10	31,13 %	68,87 %
4.s13r1b77	7,65 %	92,35 %
5.s13r1b78	31,41 %	68,59 %
6.s14r2b16	12,96 %	87,04 %
Total	109,36 %	490,64 %
Average	18,23 %	81,77 %
Maximum	31,41 %	93,68 %
Minimum	6,32 %	68,59 %
StDev	10,20 %	10,20 %
Avg/Max	0,58	0,87

Taula 20:Estat dels processadors, partitions 16

El que veiem és que l'eficiència respecte la versió inicial ha augmentat força amb valors semblants a la de l'optimització de *Master Tree* arribant gairebé als 82%, sobretot quan augmentem el nombre de particions, que no és més que dividir l'arbre en més parts i de retruc fer treballar el *master* ja que ha de llegir més vegades els arxius. Veiem també que a l'augmentar el nombre de particions estem accentuant el desbalanceig de càrrega que hi ha entre els nodes *slaves* i que tot i que augmentem el nivell d'eficiència general de l'aplicació és possible que si augmentem en gran mesura les particions al final tindrem que el desbalanceig serà tant alt que provocarà un pèrdua de rendiment general.

Pel que fa als temps d'execució, veiem que per aquest joc de proves petit no hi ha gaire diferències, des dels 4,57 segons el més ràpid (1 partició) al 4,85 segons de l'últim (16 particions). O sigui que pel que fa a temps ja tenim la gràfica anterior que ens il·lustra el comportament que aquest té en funció del nombre de particions.

En definitiva aquesta optimització creiem que ha reduït en gran importància l'ús de memòria, eliminant teòricament la limitació de memòria dels equips en compensació d'un augment del temps d'execució i com hem comentat abans pensem que és un gran avenç pel que fa als estudis poblacionals, permeten teòricament un numero il·limitat de pacients[20], i la possibilitat de executar-se en qualsevol màquina sense importar la quantitat de memòria que aquesta tingui. A més també hem aconseguit millorar l'eficiència de l'aplicació sobretot gràcies a augmentar la càrrega de treball que fa el node *master*, tot i la pèrdua de rendiment d'alguns *slaves.*

4.4.3 Reverse

Una de les dades extretes de la primera versió i que ens preocupaven era la pèrdua de *coverage* que teníem en els punts investigats, tant els llocs de mutació que havíem buscat com altres punts a l'atzar, tot mantenint el mateix nivell de profunditat que els punts de mutació per poder-los comparar. Per pèrdua de *coverage* ens referim a tenir uns comptadors molt menors en comparació el *coverage* inicial, en el cas de l'"*In silico"* de 30.

Nosaltres vam atribuir aquesta pèrdua de *coverage* a quatre factors, principalment:

- Mutació dins prefix
- *Reads* amb contaminació
- Mutació heterozigota
- 2 hèlixs

La primera és deguda a que quan un *read* conté la mutació dins els primers 30 nucleòtids (o sigui dins el prefix), llavors aquest *read*, no anirà cap al mateix arbre que els normals ja que és diferent en el prefix i per tant no estarà en el punt on busquem. Això provocarà que estarem inserint el sufix en una altra part de l'arbre però que no l'estarem utilitzant, aquests es podrien intentar trobar un cop hem analitzat una mutació i tenim la zona reconstruïda, a l'identificar el punt de mutació buscar tot els sufixes que puguin estar relacionats o sigui que siguin pròxims i buscar-los dins l'arbre amb i sense mutació.

[20] Cal mencionar que en els estudis poblacionals no s'acostuma a tenir un gran nombre de *coverage* i per tant els arxius d'un individu tenen una mida molt més petita.

La segona és quan tenim una contaminació en el procés de seqüenciació, que és un procés físic i que per tant té un marge d'error que provoca que el *read* en qüestió sigui incorrecte, tot i que treballem amb el genoma "*In silico*", aquest també té uns errors induïts per tal de que s'assembli a un genoma real seqüenciat.

L'altre possible cas és que sigui una mutació heterozigota, que és aquella mutació que només és present en un dels dos al·lels[21] del pacient. Per tant pot ser que tinguem *reads* que cobreixen una mutació d'un al·lel (que realment té la mutació) i *reads* de l'altre. Aquest punt només es produït en el cas de les zones tumorals i pot passar tant als comptadors tumorals com els normals. Malauradament no és possible corregir aquest efecte ja que es provocat pel mètode de seqüenciament i per tant inherent en els arxius d'entrada, però com que sabem que existeix podem mirar de suavitzar els filtres de comptadors per tal que no es filtrin aquest tipus de mutacions.

Per acabar a més a més, tenim el defecte que durant la seqüenciació s'agafa una de les dues hèlix que formen l'ADN indistintament i aquesta hèlix estan unides d'una manera determinada. I és que tot i ser la mateixa informació en les dues, està invertida amb el sentit que quan en una tenim la guanina (G) a l'altra tenim la citosina (C) i recíprocament, també tenim aquesta relació amb la adenina (A) i la timina(T). I a més quan la màquina selecciona l'altra hèlix el *read* es llegeix al revés. O sigui que a part d'estar capgirada a la vegada també esta invertida, si entenem inversió per el canvi de A a T (i viceversa) i de C a G (i viceversa).

Per això nosaltres vam pensar en inserir el que anomenem el *reverse*, que no és més que per cada *read*, crear un *read* revers que vol dir capgirar-lo i canviar A per T, T per A, C per G i G per C (o sigui invertint-lo). D'aquesta manera augmenten en gran quantitat el volum de dades que introduïm al programa però recuperem part del *coverage*. Els resultats van ser els següents:

Cromosoma 20	Arxius BAM (originals)		***Suffix Tree***	
	Normal	Tumor	Normal	Tumor
Sense Revers	29.79	29.98	**10.89**	**10.32**
Amb Revers	58.79	60.02	**21.22**	**21.12**

Taula 22: Valors de coverage amb revers i sense dins l'arbre

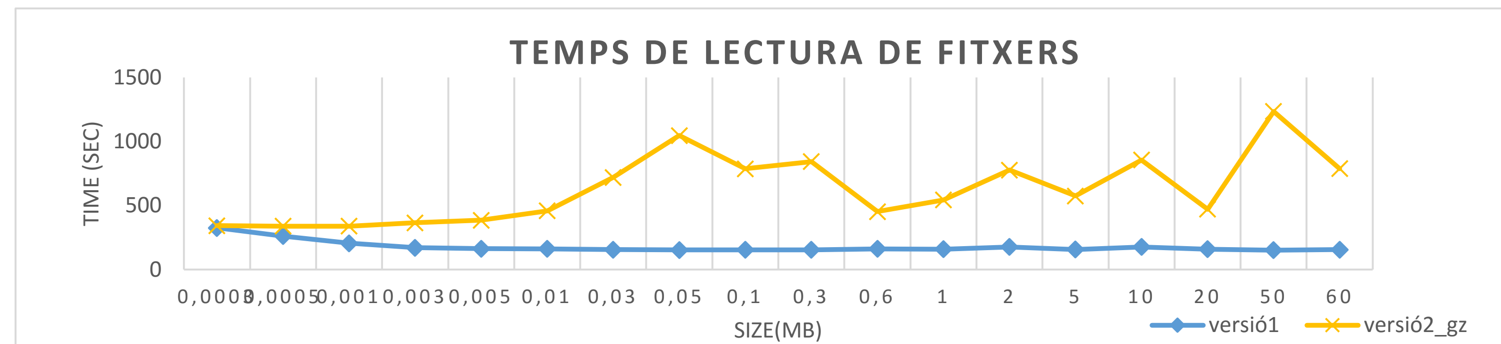
Veiem doncs que hem complert l'objectiu que teníem d'augmentar el *coverage*, ja que l'hem duplicat en tots els casos. El problema és que per això hem hagut de duplicar l'entrada del programa que ens ha fet duplicar la memòria requerida i quasi triplicar el temps d'execució. És per això que tot i assolir l'objectiu, deixarem l'opció dins el programa però estarà inhabilitada per defecte.

[21] Cadascuna de les formes alternatives que pot presentar un gen que ocupa el mateix lloc en un cromosoma determinat o en dos cromosomes homòlegs, i que expressa diferentment un mateix caràcter.

4.4.4 Lectura d'arxius comprimits

És molt comú en aquest camp, degut a que els arxius d'entrada són de grans dimensions, que estiguin comprimits és per això que vàrem creure necessari la característica de poder llegir arxius d'aquest tipus, en concret comprimits amb *Zip* (amb extensió per exemple .gz).

Així que vam crear una nova funció de lectura de fitxers fastq totalment renovada que ens permetia llegir els arxius comprimits directament sense necessitat de descomprimir-los tots sencers, sinó utilitzant l'estratègia que havíem emprat anteriorment d'utilitzar el *buffer* de lectura. Com que aquest algoritme era més lent vàrem decidir de mantenir l'algoritme antic i utilitzar aquest nou en el cas només dels arxius comprimits. A continuació veiem una gràfica de temps de la lectura dels fitxers en funció de les mides de *readblock*, tant l'algoritme inicial com el nou, això si, el nou utilitza arxius comprimits:



Il·lustració 25: Gràfica de temps de la lectura de fitxers en les diferents versions

Com hem comentat el rendiment del nou algoritme és pitjor, val a dir però que tracta amb arxius comprimits i que si ho volguéssim fer amb la versió inicial hauríem de descomprimir les dades prèviament a l'execució i per tant tot i que sembli pitjor de moment és l'única opció que tenim.

4.4.5 Comunicació *Master-Slaves*

La versió inicial té una versió molt bàsica pel que fa a comunicació entre nodes és per això que vam decidir de millorar aquest apartat, per a fer això vam pensar en dues crides diferents:

MPI_Ibcast

Vam estar desenvolupant un versió que utilitzava la crida *MPI_Ibcast,* que és fer un *Broadcast,* però de manera asíncrona, la complexitat alhora de programar es bastant alta perquè és necessari crear i administrar un *buffer* de sortida que s'actualitzi a mesura que o bé el *master* acaba un bloc de lectura o bé tots els nodes han informat que s'ha llegit un bloc i per tant es pot eliminar. Tot i que vam arribar a implementar una versió inicial que funcionava amb arxius de mida reduïda, no vam ser capaços de poder-la utilitzar en arxius més grans i a més la poca informació que vàrem trobar de la funció ens va fer sospitar que no devia ser tant bona respecte a la seva homòloga síncrona.

Per altra banda actualment estem treballant amb una versió que utilitza la funció *MPI_Bcast* per a realitzar les comunicacions entre els nodes ja que és molt més eficient en quant a temps i ens permetria millorar per tant el rendiment de l'aplicació. La dificultat que aporta aquesta funció és que tots els nodes tenen que realitzar la crida en el mateix punt i per tant cal fer una fusió de funcions de *master* i *slaves* per tal que puguin arribar a fer-la.

4.5 Resultats

Com hem comentat abans per tal de capturar resultats hem hagut d'implementar uns filtres amb l'ajuda dels companys del grup multidisciplinari, per això a continuació presentarem alguns resultats amb alguns d'aquests filtres i combinacions d'alguns d'ells, recordem que estem utilitzant l'estratègia anomenada *Tree Oriented*. Per començar presentarem les dades del cromosoma amb el què hem treballat:

Cromosoma 20	
Point Mutations:	168
Inversions	6
Deletions	12
Insertions	18

Taula 23: Mutacions en el cromosoma 20

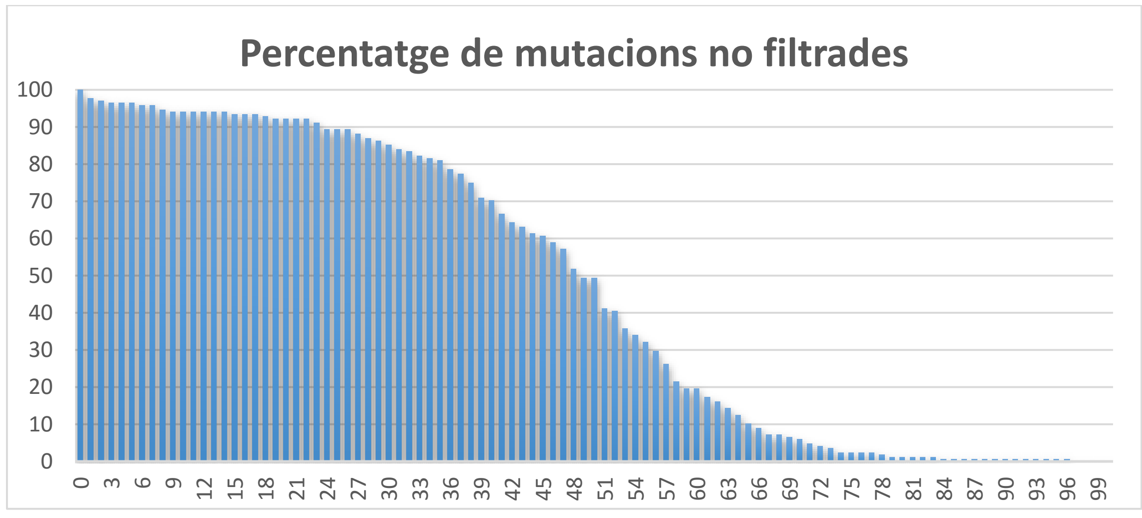
Degut al gran nombre de les *Point Mutations* ens hem decidit d'extreure els resultats d'aquestes ja que en els altres casos el nombre és massa petit per tal d'intentar extreure algun patró que els diferenciï. Les *Point Mutations* és la mutació més simple, es tracta del canvi d'un sol nucleòtid, ja sigui perquè ha canviat a un altre nucleòtid, perquè se n'ha eliminat un o perquè se n'ha afegit un de més.

El primer filtre que vam dissenyar és el següent:

$$\text{valor} \leq \frac{100{*}T_i}{T_A + T_C + T_G + T_T}$$

On T_i és el comptador tumoral en un punt i T_A, T_C ... són els comptadors tumorals dels fills.

El resultat que teníem en funció del valor del filtratge és el següent:

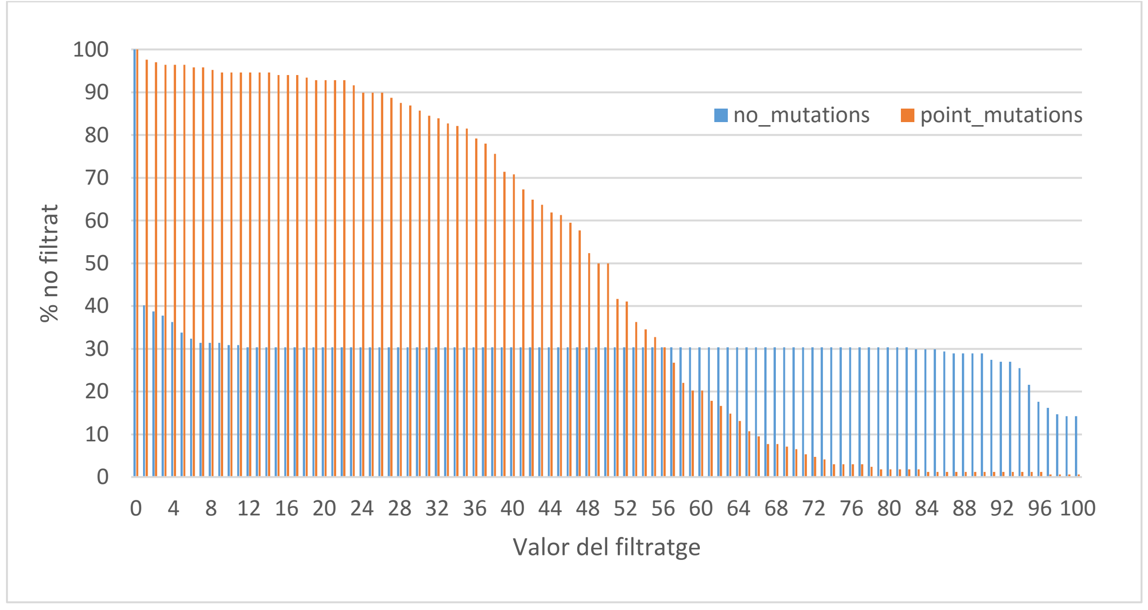


Il·lustració 26: Resultat d'aplicar el 1r filtre

Com veiem el nombre de mutacions que filtrem augmenta a mesura que el filtre és més rigorós, com per altra banda era previsible, d'aquesta manera veiem que el valor òptim estaria entre 2 i

5. El problema que se'ns presenta és que si escollim un nivell tant baix és molt probable que el nombre de nodes que passaran el filtre serà tant gran que realment no haurà tingut molt impacte l'aplicació d'aquest filtre.

Per això vam voler repetir aquest filtre amb aquell conjunt de punts aleatoris que no contenien mutació per tal de veure quin era el "gruix" de punts no tumorals que realment filtràvem amb aquest filtre. El comportament és el següent:



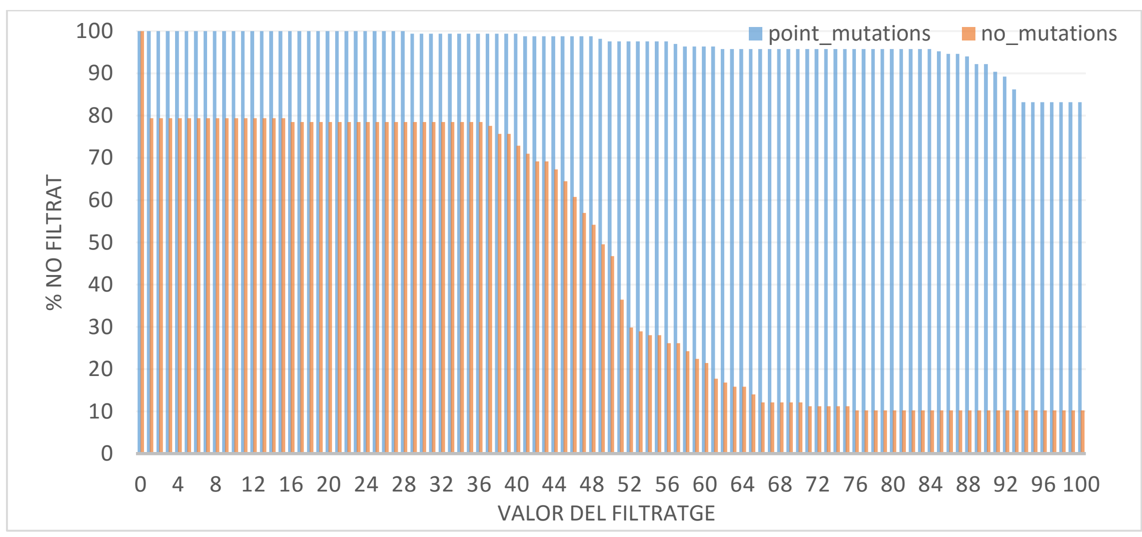
Il·lustració 27: Resultat d'aplicar el 1r filtre en els dos tipus de posicions

Veiem doncs que en realitat si que estem filtrant els nodes naturals, per exemple si agafem el valor de 20, estem mantenint un 92,16 % de les mutacions i en canvi dels punts de no mutacions els estem reduint al 30%. Tot i així com hem vist abans si establim el filtre a 20 estem perdent un nombre de mutacions massa elevat pels objectius que ens vam marcar. És per això que en comptes d'aplicar aquest filtre amb el valor de 20 vam decidir d'aplicar-lo a 4 i buscar algun altre filtre que solucionés el volum de punts de no mutació.

El filtre que vam pensar per eliminar els falsos punts de mutació va ser el següent:

$$valor \leq \frac{100 * T_i}{N_i + T_i}$$

Degut a que en els punts de mutació un dels seus fills conté un gran nombre de tumorals en comparació als normals. El filtre no elimina els punts en què un dels seus fills compleix la desigualtat anterior. D'aquesta manera obtenim el següent resultat:



Il·lustració 28: Resultat d'aplicar el 2n filtre en els dos tipus de punts

Com veiem el resultat del 2n filtre és un èxit ja que com veiem en la il·lustració anterior ens permet filtrar en gran mesura els punts de no mutació i no perdem un gran nombre de mutacions, veiem doncs per exemple el punt de 60, en què mantenim un 96,41% de les mutacions i reduïm els punts de no mutació fins a un 21’5%.

4.6 Estudis poblacionals

Per tal d’elaborar la versió d’estudis poblacionals vam partir del codi de la versió somàtica amb les optimitzacions abans descrites. La principal restricció que ens vam trobar va ser la limitació de temps de manera que les solucions proposades queden lluny del nivell d’optimització que hauríem desitjat que tinguessin.

El principal problema que se’ns planteja en aquest apartat és que ara no construirem un arbre a partir de dos genomes i comparant diferències un amb l’altre sinó que a part de que tindrem un nombre més elevat de genomes, aquests hauran de comparar-se tots amb tots dins l’arbre. La manera en que vam idear per solucionar aquest problema va ser la de no utilitzar els comptadors tumorals dels nodes i només utilitzar els comptadors normals. Per tal d’identificar en cada node a quin pacient pertanyien els sufixes vam idear una taula que ens indiqués els identificadors (*ids*) per a cada pacient com la que veiem a continuació:

patient0	0	10637
patient1	10638	11795
patient2	11796	14193
patient3	14194	14745
patient4	14746	15159
patient0	1	10638
patient1	10639	11796
patient2	11797	14194
patient3	14195	14746
patient4	14747	15160

Il·lustració 29: Taula d'identificadors dels pacients

Amb l'ajuda d'aquesta taula i la creació d'una funció que ens retorna tots els identificadors que hi ha a l'interior d'un node va ser suficient per tal d'incorporar diferents pacients i poder-los identificar. Per tal de validar que la informació que conté l'arbre és correcta vam utilitzar l'estratègia *Tree Oriented*, aquesta vegada però els filtres aplicats van ser diferents. La idea era obtenir pocs resultats per tal de poder-los verificar buscant per punts a l'atzar aquell mateix sufix als arxius d'entrada i corroborar doncs que els reads que ens mostrin els resultats són els correctes. Els arxius d'entrada utilitzats són els cromosomes 20 corresponents a 5 pacients aleatoris obtinguts de la pàgina mil genomes.

Seguidament detallarem els dos filtres utilitzats:

El primer que vam aplicar va ser que el node tingués com a mínim 60 sufixes, d'aquesta manera reduíem en gran mesura els nombre de punts.

El segon filtre consistia en que els punts continguessin sufixes de tots els pacients, així veuríem punts en el que es poguessin veure les diferents variacions genètiques en la població, aquesta es veuria en la diversificació dels punts trobats en els seus fills. Per exemple tenir comptadors de 60 al node trobat i veure com 2 pacients tenen com a fill una C mentre que els altres tres tenen una A.

5 Conclusions

5.1 Conclusions generals

El principal objectiu del projecte era la creació d'una aplicació paral·lela, eficient i escalable que faria de pont entre els camps de la biologia i la informàtica. El fet d'estar entre dos camps tant diferents va requerir un estudi previ en ambdós casos però sobretot entendre el context biològic sobre el qual es desenvolupava. A més també va ser necessàri un estudi de l'estructura *Suffix Tree* ja que aquesta és la base de l'estratègia que es volia emprar.

Per tal de comprendre l'apartat biològic van ser necessàries a part de la lectura de diferents articles relacionats, les explicacions de membres del grup de treball pertanyents a aquesta matèria. Posant èmfasis als articles val destacar el de SMUFIN[3], que servia per entendre l'estratègia del programa i que ja si mesclava l'estructura principal (*Suffix Tree*) amb els termes biològics pertanyents.

Pel que fa referència al camp informàtic, i més en profunditat al de Big Data, es van llegir també alguns articles i ens vam fixar inicialment en el paradigma de Map Reduce, amb el qual es va començar a implementar una versió molt senzilla que acabaria sent descartada i substituïda amb la utilització de la tecnologia MPI com a comunicació i estructuració dels nodes.

Un cop ja escollides les tecnologies que s'emprarien i entenent el rerefons biològic es va anar desenvolupant l'aplicació fins a aconseguir una versió funcional bàsica. A partir d'aquí ens vam interessar principalment en verificar el correcte funcionament d'aquesta, i un cop validada, se'ns van obrir diversos fulls de ruta a seguir.

El primer d'ells era buscar uns filtres adequats, per a que en la pràctica pugui ser emprada obtenint com a resultat el màxim de mutacions possibles i reduir al màxim els possibles sorolls (o falsos positius), millorant d'aquesta forma la sensibilitat i l'especificitat de l'aplicació.

Seguint l'objectiu de millorar l'aplicació teníem també la de millorar-ne l'eficiència en quant als recursos que aquesta emprava, en aquest sentit vam dissenyar un anàlisi del funcionament de l'aplicació amb la utilització d'eines que ens permetien instrumentar el codi i veure'n el comportament d'aquest durant l'execució almenys per a jocs de proves petits. El resultat han set un conjunt d'optimitzacions que en ocasions no han tingut el resultat esperat amb un gran volum de dades però en altres casos, com és el cas de *Partitions*, ens han portat un gran benefici. L'optimització *Partitions* es basa en modificar el funcionament general de l'aplicació gràcies a dividir la construcció de l'arbre (*Suffix Tree*) en fases. D'aquesta manera ens permet reduir la memòria requerida per l'aplicació i per tant fer-la més portable per a altres equips que tinguin menys memòria a mesura que augmentem el nombre de fases que aquesta té.

Per acabar també ens havíem marcat l'objectiu de crear una implementació vàlida per al cas dels estudis poblacionals per tal de validar l'ús de l'estructura del *Suffix Tree* també en aquest tipus d'estudis. En aquest apartat cal destacar que hem tingut un obstacle afegit degut al poc temps disponible que ens restava. Com a conseqüència aquesta implementació ha set molt senzilla i poc òptima. Com a resultat però hem pogut establir un arbre amb el cromosoma 20 de 5 pacients diferents i fer unes validacions inicials per tant aconseguint l'objectiu inicial.

Per acabar comentar que tot i només haver provat l'aplicació en diferents cromosomes, la validació en aquests és suficient com per garantir l'execució en genomes sencers.

5.2 Ètica del projecte i regulacions adherides

Aquest projecte està regulat per la clàusula de confidencialitat i de propietat intel·lectual amb l'empresa on es realitza, el BSC (Barcelona Supercomputing Center).

Per altra banda les dades que s'han utilitzat en el curs del projecte, provenen en gran part d'un codi genètic creat artificialment (anomenat "In silico"). En canvi les utilitzades en el cas d'estudi poblacional provinents de 5 pacients han set extretes d'una base de dades que guarda amb total anonimat qualsevol dada que pugui servir per identificar el pacient a part del genoma, i que per altra banda garanteix que van ser extrets de forma voluntària. Per tant l'ús d'aquestes dades no hauria de suposar cap dilema ètic.

5.3 Sostenibilitat i compromís social

Per a poder realitzar un estudi sobre la sostenibilitat del projecte, es puntuarà des de tres punts de vista diferents: econòmic, social i ambiental. Per a cada una d'aquestes dimensions s'utilitzarà el mètode socràtic per obtenir una puntuació, les preguntes són les utilitzades a l'assignatura GEP i per avaluar-les s'utilitzarà una matriu que ens proporcionen els professors de GEP.

5.3.1 Dimensió econòmica

En aquesta dimensió la planificació del projecte és puntuat amb un 8.3. Justificacions:

- Es fa una avaluació exhaustiva dels costos materials i humans del projecte: 9
- S'han tingut en compte desajustos i actualitzacions i manteniments del MareNostrum III : 8
- Tot i que molts dels recursos són de cost zero el cost del projecte és elevat: 6
- El projecte no es podria reduir molt de temps ja que totes les tasques són seqüencials i com que és un camp de recerca i s'intenta no utilitzar llibreries externes s'han de fer continues: 8
- Les tasques més importants són les de desenvolupar les aplicacions, les altres tasques com per exemple les documentacions prèvies tenen un temps assignat molt menor: 9
- Aquest projecte està dins d'un projecte de recerca molt més gran que s'està duent a terme al departament de *Life Science* i en concret al grup de *Computational Genomics* del BSC: 10

5.3.2 Dimensió social

La puntuació que ha tret en aquest apartat és de 8.83. Justificació:

- La situació social del país és benestant, i la política és estable i democràtica: 10
- La millora del programa SMUFIN com a conseqüència de l'èxit d'aquest projecte, pot impulsar la medicina personal i la millora en les deteccions de l'origen de diverses malalties genètiques: 9
- La necessitat existeix ja que les morts en el 1r món degut a malalties genètiques són de les més elevades: 9

- Com que la medicina avançarà en certa manera i a la llarga, com a conseqüència del projecte SMUFIN, es pot considerar una millora de la qualitat de vida: 8
- No es pot saber com afectarà el resultat de l'usuari, però segur que no perjudicarà en res: 7
- No hi ha cap col·lectiu que es vegi perjudicat pel projecte: 10

5.3.3 Dimensió ambiental

En aquesta dimensió la planificació del projecte obté un resultat de 7.75. Justificació:

- L'únic recurs que consumeix que pot afectar al medi ambient és la llum. I no l'afecta directament, sinó indirectament, segons com ha estat produïda aquesta energia elèctrica (fonts renovables o no renovables): 8
- Durant la fase de desenvolupament i posada en marxa, l'ordinador personal tindrà un consum molt baix de llum. En canvi, el MareNostrum III ja té un consum molt més elevat, tot i que s'ha de tenir en compte que s'hi executen moltes altres tasques, per tant, només s'ha d'atribuir una petita part a aquest projecte: 8
- A efectes teòrics, aquest projecte disminueix el consum d'energia del model del BSC. Però a efectes pràctics, el MareNostrum III sempre està en funcionament, per tant, no disminuirà el consum de llum: 6
- S'ha reutilitzat parts d'altres projectes com són el codi del *Suffix Tree*: 8
- Només produirà contaminació indirectament, és a dir, en la producció de la llum que consumeix: 7
- No s'usarà cap matèria primera per a realitzar el projecte: 9
- Molt probablement no disminueixi l'emprempta ecològica, però segur que no l'augmentarà: 7
- Aquest projecte beneficiarà directament el projecte SMUFIN ja que bàsicament aquest projecte es fa amb l'objectiu de millorar-ne l'eficiència i les dependències externes: 9

Per acabar podem veure a la següent taula el resum dels 3 tipus de sostenibilitat i la puntuació final obtinguda en cada una d'elles:

Sostenibilitat	*Econòmica*	*Social*	*Ambiental*
Planificació	Viabilitat econòmica	Millora en qualitat de vida	Anàlisi de recursos
Valoració	8.3	8.83	7.75

Taula 24: Resum de les puntuacions obtingues en Sostenibilitat

6 Referències

1 – Projecte UK10K consultada el 6 d'agost de 2015, a http://www.uk10k.org/

2 – Projecte GoNL (Genome of the Netherlands) consultada el 6 d'agost de 2015, a http://www.nlgenome.nl/

3 – Moncunill V., Gonzalez .S, Beà .S, O Andrieux .L, Salaverria I., Royo C., Martinez L., Puiggròs M., Segura-Wang M., M Stütz A., Navarro A., Royo R., Gelpí JL., G Gut I., López-Otín C., Orozco M., O Korbel J., Campo E., S Puente X. & Torrents D..

The Health Sience Dpt/BSC-CNC: Computational Genomics.

Comprehensive characterization of complex estructural variatons in cancer by directly comparing genome sequence reads

Nature (2014) Enllaç: http://www.nature.com/nbt/journal/v32/n11/full/nbt.3027.html (Juliol 2015)

4 - Ye K, Schulz MH, Long Q, Apweiler R, Ning Z. Pindel: *a pattern growth approach to detect break points of large deletions and medium sized insertions from paired-end short reads*. Bioinformatics. Novembre del 2009 1;25(21):2865-71. Epub 2009 Jun 26. Enllaç consultat el 4 d'agost de 2015: http://gmt.genome.wustl.edu/packages/pindel/index.html

5 - Ken Chen, John W Wallis, Michael D McLellan, David E Larson, Joelle M Kalicki, Craig S Pohl, Sean D McGrath, Michael C Wendl, Qunyuan Zhang, Devin P Locke, Xiaoqi Shi, Robert S Fulton, Timothy J Ley, Richard K Wilson, Li Ding & Elaine R Mardis: *BreakDancer: an algorithm for high-resolution mapping of genomic structural variation.* 9 d'agost de 2009, *Nature* Enllaç consultat el 4 d'agost de 2015: http://breakdancer.sourceforge.net/

6 - Handsaker RE, Van Doren V, Berman JR, Genovese G, Kashin S, Boettger LM, McCarroll SA: *Large multiallelic copy number variations in humans. Nature Genetics 47* (GenomeSTRIP) Enllaç consultat el 4 d'agost de 2015: http://www.broadinstitute.org/software/genomestrip/

7 – GATK (Genome Analysis Toolkit, Broad Institute) Enllaç consultat el 4 d'agost de 2015: https://www.broadinstitute.org/gatk/

8 – CREST (Clipping Reveals Structure) Enllaç consultat el 4 d'agost de 2015 , actualment caigut: http://www.stjuderesearch.org/site/lab/zhang

9 - Tobias Rausch, Thomas Zichner, Andreas Schlattl, Adrian M. Stuetz, Vladimir Benes, Jan O. Korbel. *Delly: structural variant discovery by integrated paired-end and split-read analysis.* Bioinformatics 2012 28: i333-i339. Enllaç consultat el 7 d'agost de 2015: https://github.com/tobiasrausch/delly

10 – Chapman, M.A., et al. (2011) *Iniitial genome sequencing and analysis of múltiple mieloma, Nature, 417, 467-472* Enllaç consultat el 5 d'agost de 2015: https://www.broadinstitute.org/cancer/cga/mutect

11 – Benjamin J Kelly, James R Fitch, Yangqiu Hu, Donald J Corsmeier, Huachun Zhong, Amy N Wetzel, Russell D Nordquist, David L Newsom i Peter White (Genome Biology 2015) : "*an ultra-fast, deterministic, highly scalable and balanced parallelization strategy for the discovery of human genetic variation in clinical and population-scale genomics*". Enllaç consultat el 6 d'agost de 2015: http://www.genomebiology.com/2015/16/1/6

12 - A. K. Lancaster, R. M. Single, O. D. Solberg, M. P. Nelson and G. Thomson (2007) "*PyPop update - a software pipeline for large-scale multilocus population genomics*" Tissue Antigens 69 (s1), 192-197

13 - Bastian Pfeifer, Ulrich Wittelsbuerger, Heng Li, Bob Handsaker : PopGenome: "*An Efficient Swiss Army Knife for Population Genomic Analyses*" Enllaç consultat el 7 d'agost de 2015 : https://cran.r-project.org/web/packages/PopGenome/index.html

14 - J. Catchen, P. Hohenlohe, S. Bassham, A. Amores, and W. Cresko. :"*Stacks: an analysis tool set for population genomics*". Molecular Ecology. 2013.

15 - Handsaker RE, Korn JM, Nemesh J, McCarroll SA :"*Discovery and genotyping of genome structural polymorphism by sequencing on a population scale*" Nature Genetics 43, 269-276 (2011)

16 – K. Menon, R. , P. Bhat, G. , i C. Schatz, M. : *Rapid Parallel Genome Indexing with MapReduce* Enllaç consultat el 7 de març de 2015: http://schatzlab.cshl.edu/publications/2011-GenomeIndexingMapReduce.pdf

17 – Bieganski P., Ned J. , V. Cadis J., E Retzel E. :*Generalized Suffix Trees for Biological Sequence Data: Applications and Implementation*: Enllaç consultat el 7 de març de 2015 http://files.grouplens.org/papers/Riedl-GeneralizedSuffixTrees.PDF

18 - Manber, Udi; Myers, Gene (1990). *Suffix arrays: a new method for on-line string searches*. First Annual ACM-SIAM Symposium on Discrete Algorithms. pp. 319–327

19 – "Lenguajes de alto nivel". A Wikipedia en el següent enllaç consultat el 8 de març del 2015 : https://es.wikipedia.org/wiki/Lenguaje_de_alto_nivel

20 – Welch, N. : *Hash Table Benchmarks* Enllaç consultat el 10 d'agost de 2015: http://incise.org/hash-table-benchmarks.html