2123 PROGRAMACIÓ D'APLICACIONS BIOINFORMÀTIQUES

Memòria del Projecte Fi de Carrera
d'Enginyeria en Informàtica
realitzat per
Josep Maria Bosch Muntal
i dirigit per
Porfidio Hernández Budé

Bellaterra, 15 de Juny de 2010

El sotasignat, ...

Professor/a de l'Escola d'Enginyeria de la UAB,

CERTIFICA:

Que el treball a què correspon aquesta memòria ha estat realitzat sota la seva direcció per en

I per tal que consti firma la present.

Signat: .....

Bellaterra, ...de.....de 20.....

TAULA DE CONTINGUTS

Capitol 1 Formulació del problema i objectius

Aquest primer capítol ha de servir a mode d'introducció per al lector. En ell expliquem el contingut del projecte i hi donem les explicacions que s'han considerat necessàries sobre el tema tractat. També hi trobem la planificació seguida i els objectius.

1.1 Organització de la memòria

La memòria s'organitza en sis capítols amb els apartats necessaris per estructurar el seu contingut. En aquest capítol introduïm el problema a resoldre, els objectius que perseguim, la planificació proposada i els aspectes més rellevants sobre la viabilitat i els costos. Al segon capítol donem la informació necessària sobre la teoria dels temes tractats i sobre les eines utilitzades. El tercer capítol expliquem el disseny i implementació de les respostes al problema plantejat. Al quart donem la experimentació realitzada i els resultats. Al cinquè capítol el dediquem a les conclusions. Finalment el sisè conté la bibliografia.

Al final del projecte annexem explicacions de com usar els paradigmes de programació tractats i els codis basics dels algoritmes analitzats per a poder ser executats com a exemple.

1.2 Formulació del problema

El títol del projecte "**Programació d'aplicacions bioinformàtiques**" deixa un ventall molt ampli de possibles temes a tractar. Així que ens proposem, en aquest primer apartat, centrar l'atenció en el problema que volem resoldre.

Per entendre quins problemes hem de resoldre a l'hora de programar-ne la seva execució primer és convenient definir a que ens referim quan parlem d'aplicacions bioinformàtiques,. El **NCBI** (**National Center for Biotechnology Information**) **[1]** engloba en la seva definició de bioinformàtica la confluència de diferents disciplines amb l'objectiu d'extreure coneixement biològic a partir del tractament, interpretació i anàlisi informàtic de les dades disponibles, així com el disseny de mètodes amb els que poder relacionar-les.

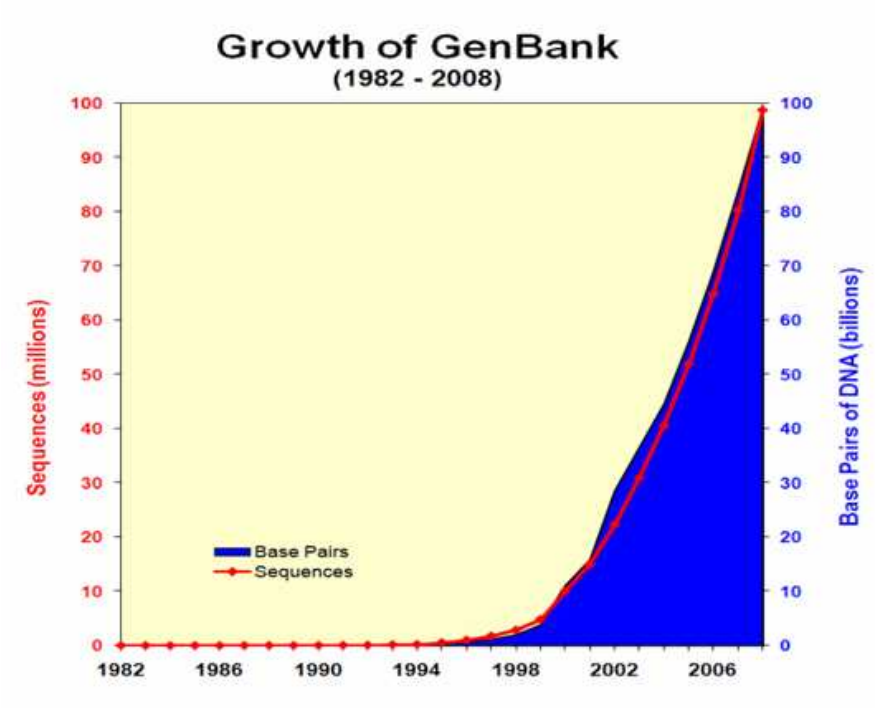

Figura 1-1. Creixement de les bases de dades biològiques fins al 2006

A la **[Figura 1-1]** s'aprecia el creixement exponencial de les dades biològiques emmagatzemades. Un dels reptes més importants de la biologia és, curiosament, entendre l'enorme quantitat de dades de que es disposa. En aquest sentit es treballa per entendre la biologia dels organismes a partir de la informació genètica. Els majors esforços d'investigació en aquest camp inclouen l'alineament de seqüències, la predicció de gens, muntatge de genomes, alineaments estructurals de proteïnes, prediccions estructurals i d'expressió i modelat de l'evolució.

Entre les raons que donen rellevància a la programació d'aplicacions bioinformàtiques estan en primer lloc el canvi en la manera en que s'analitza la informació biològica. Aquest passa del tractament tradicional de laboratori als mètodes més econòmics i eficients basats en el còmput informàtic i l'ús de sistemes paral·lels. En segon lloc està la necessitat de tractar amb total detall les dades biològiques ja que el més petit canvi a nivell molecular pot afectar a totes les cèl·lules, teixits i òrgans.

1.3 Introducció a les aplicacions bioinformàtiques

En aquest projecte ens centrem en la part de la bioinformàtica que s'ocupa de trobar mètodes per tractar dades biològiques a fi d'extreure'n informació, i més concretament de l'anàlisi d'aplicacions per a alineament de seqüències.

La informació biològica a que ens referim és en general l'expressió genètica dels essers vius o més concretament l'**ADN**.

L'**ADN** conte la informació genètica usada en el desenvolupament i funcionament dels organismes vius i és el responsable de la transmissió hereditària. Està format per moltes bases nitrogenades connectades entre si amb una unitat de desoxiribosa (un sucre) i un grup fosfat. Dins d'una cèl·lula com la representada a la **[Figura 1-1]** l'**ADN** està formant cromosomes al nucli.

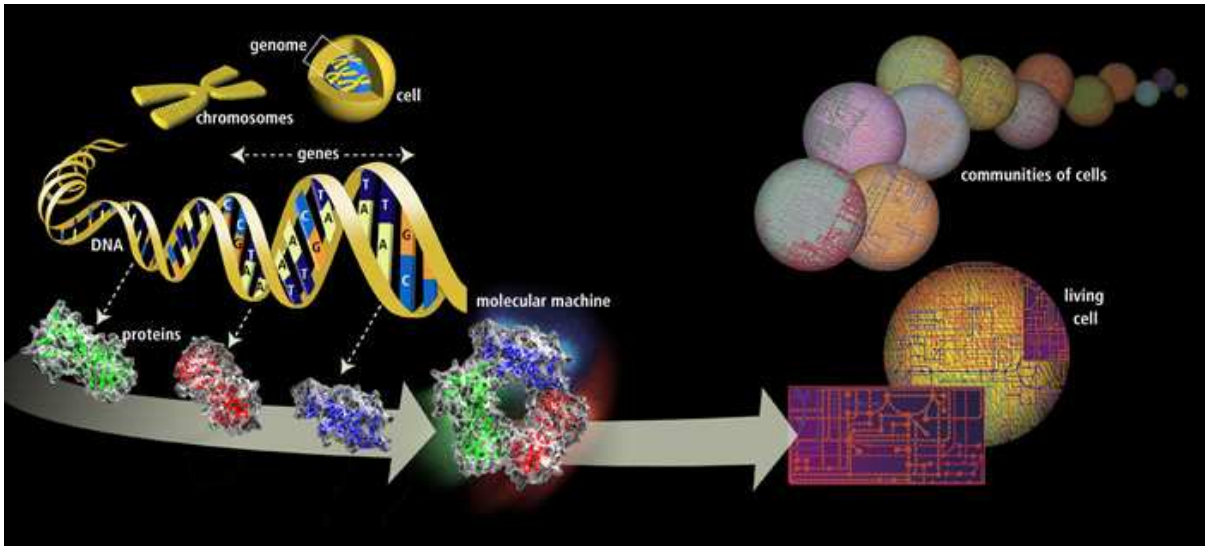

Figura 1-1.Representació de la cèl·lula animal

La informació genètica es copia en el procés de transcripció desenvolupat al nucli creant **ARN**. Aquest es desplaça al citoplasma on s'utilitza per construir proteïnes i altres òrgans cel·lulars en el procés de traducció.

Representem l'**ADN** en seqüències de lletres que representen les quatre bases nitrogenades que la formen.

L'alineament de seqüències ens permet obtenir informació d'una cadena de dades biològiques per trobar-ne semblances a una altra de coneguda. Això ens permet fer prediccions i inferències que ajuden a entendre aquestes dades. **Alinear** significa comparar dues o més seqüències per a reconèixer els punts de similitud o donar un valor de semblança segons una mètrica donada (**score**).

Els algoritmes que estudiarem estan basats en programació dinàmica, o sigui a partir d'una discreció i de forma seqüencial, i són òptims (donen el millor resultat possible). Aquests algoritmes, en la seva definició, tenen unes complexitats d'**ordre n^2**, tant en el còmput com per a les necessitats de memòria. La necessitat de treballar amb problemes grans amb complexitat

d'ordres quadràtics ens fa pensar en la conveniència d'analitzar com podem implementar aquests algoritmes en paral·lel i analitzar-ne els resultats.

1.4 Objectius del projecte

L'objectiu general del projecte consisteix en analitzar la problemàtica que suposa l'alineament de parells de seqüències, tant a nivell global com local, i les estratègies clàssiques per resoldre-ho.

Ens centrem per al anàlisi amb els algoritmes **Needleman&Wunsch**, per l'alineament global, i **Smith&Waterman**, per al alineament local; tots dos donen resultats òptims.

Proposarem diferents implementacions i les compararem amb la finalitat de millorar els resultats en temps i memòria requerits per a l'anàlisi d'alineaments de seqüències grans.

Donats els següents antecedents:

- Avui en dia es disposa de grans bases de dades, amb enormes quantitats d'informació genètica de la que queda molt per entendre. Aquesta s'ha obtingut amb el que es coneix com a tècniques biològiques d'alta productivitat i s'ha de poder comparar massivament per a poder extreure'n informació útil.
- Per a tractar la massiva informació es disposa, amb facilitat, de grans clústers on executar algoritmes en paral·lel. Així podem plantejar-nos treballar amb volums de dades cada cop més grans o afrontar problemes de magnituds molt superiors. Això sí, sempre que el problema es pugui resoldre en paral·lel.
- Analitzant els consums de temps i memòria dels algoritmes clàssics, basats en programació dinàmica i òptims, per a grans volums de dades, s'arriba a la necessitat de plantejar-ne versions paral·lelitzables, així com l'ús de nous paradigmes de programació, per a tractar els problema de cara a la manipulació de grans volums de dades.
- Els algoritmes amb que treballem tenen complexitats temporals i espacials d'ordres **n*m**, on **n** i **m** són les mides de les entrades.

Volem provar la aplicabilitat de nous paradigmes de programació per resoldre l'alineament òptim entre parells de seqüències, i comparar-ne el rendiment.

En primer lloc es vol plantejar la paral·lelització dels algoritmes per a implementar-ne la solució usant diferents paradigmes. En aquest sentit es pensava en **MapReduce**, un paradigma de

programació usat per **Google** per a la programació paral·lela, el qual es vol comparar amb altres alternatives per a la programació paral·lela com **MPI** (**Message Passing Interface**).

Per a les diferents versions dels algoritmes implementats se n'han de prendre mesures de rendiment per comparar-les.

També esperem poder arribar a treure conclusions de les dades preses i les comparatives. Aquestes conclusions volem encaminar-les a valorar la millora de rendiments, l'escalabilitat i paral·lelització del problema i els nous ordres de complexitat obtinguts.

Finalment hauríem d'arribar a proposar maneres de solucionar el problema que poguessin millorar les donades, sempre pensant en donar els millors resultats per a l'alineament de seqüències amb un nombre de bases el més grans possibles.

1.5 Metodologia i planificació de les tasques

Al abordar el projecte seguim una metodologia cíclica, aquesta vol millorar els resultats del problema abordat a cada iteració. Des de l'anàlisi del problema inicial i la implementació del algoritme en **C++**, s'analitzen els resultats obtinguts i se'n detecten els punts a millorar per a proposar una nova implementació.

Així seguim els següents passos, basats en el **mètode del cicle de vida clàssic [14]** de forma iterativa:

1. Anàlisi del problema
2. Proposta de solució
3. Anàlisi i test de la solució
4. Detecció de punts on millorar

Del seguiment d'aquest esquema, i basant-nos en les limitacions a cada pas, en surten les diferents implementacions per al càlcul dels alineaments seguint els algoritmes **Needleman&Wunsch** (en endavant **N&W**) i **Smith&Waterman** (en endavant **S&W**):

- Versió que fa els càlculs en sèrie
- Versió usant **Threads**
- Versió usant **MPI**
- Anàlisi de la solució usant **MapReduce**

A més s'analitza el problema i la possibilitat d'usar el paradigma **MapReduce** per a la implementació dels mateixos algoritmes. Per a dur a terme totes les tasques es té en compte la planificació representada a la **[Figura 1-2]**.

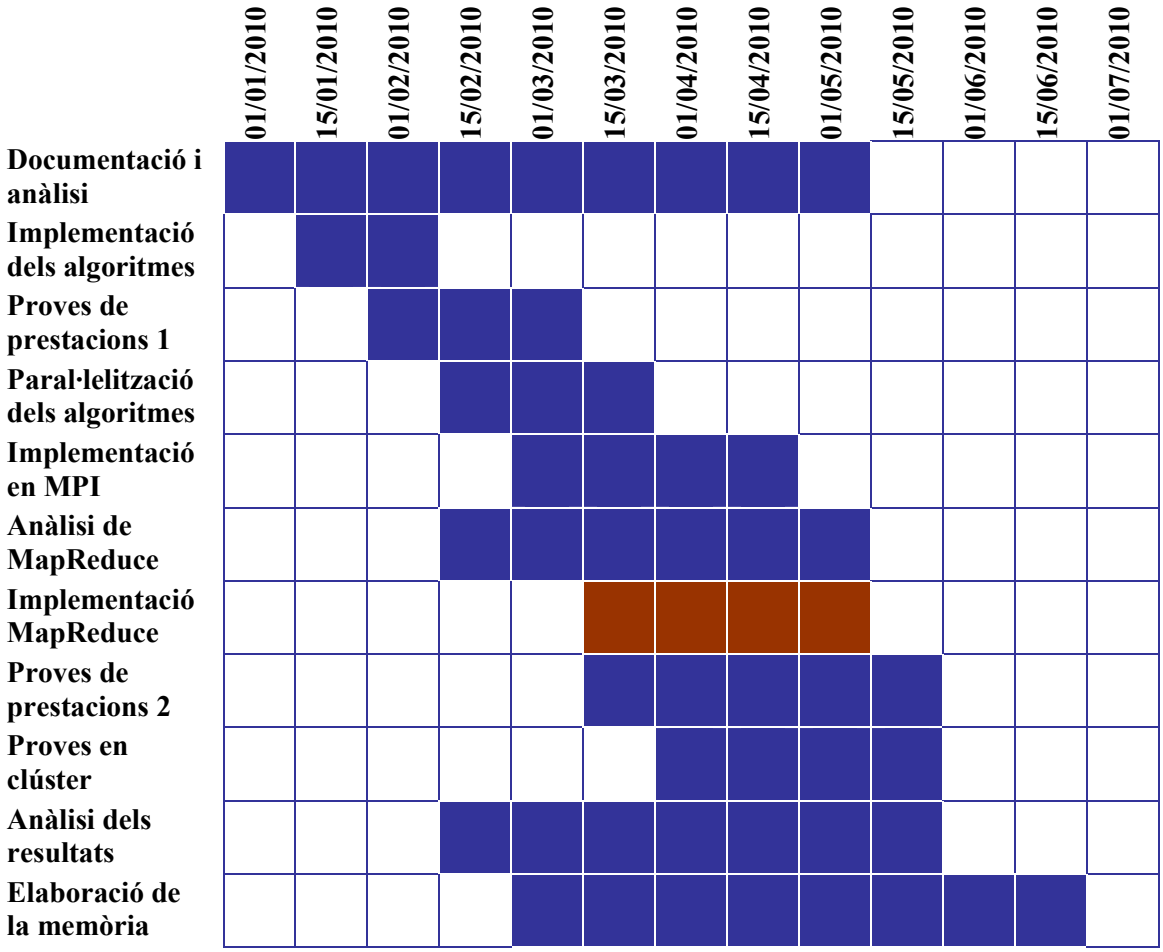

Figura 1-2. Planificació inicial de les tasques

1.6 Viabilitat del projecte

En la planificació proposada esperem poder aconseguir implementacions paral·leles de les aplicacions bioinformàtiques estudiades. La viabilitat del treball està condicionada per l'èxit en l'anàlisi dels algoritmes.

A partir de les versions paral·leles dels algoritmes no podem garantir que els resultats siguin una millora significativa del problema tractat en quan a complexitat espacial i temporal.

Per a provar les implementacions necessitem disposar de un mínim de tres ordinadors on executar els programes en paral·lel. El software necessari pot ser obtingut gratuïtament en versions lliures.

Capitol 2 Fonaments teòrics

Introduïm aquí tota la base teòrica que considerem d'interès de cara a justificar les solucions i els plantejaments proposats al llarg del treball. Entre la informació inclosa hi ha la base sobre els algoritmes d'alineament en que es basa l'estudi posterior. També hi hem inclòs les especificacions sobre les màquines que s'han usat i els paradigmes amb els que hem treballat.

2.1 Algoritmes d'alineament de seqüències

Com hem vist al primer capítol i seguint l'esquema de la **[Figura 2-1]** les seqüències d'**ADN** són una representació dels nucleòtids que en formen les estructures representats a partir de quatre lletres, **A**, **C**, **G** i **T**:

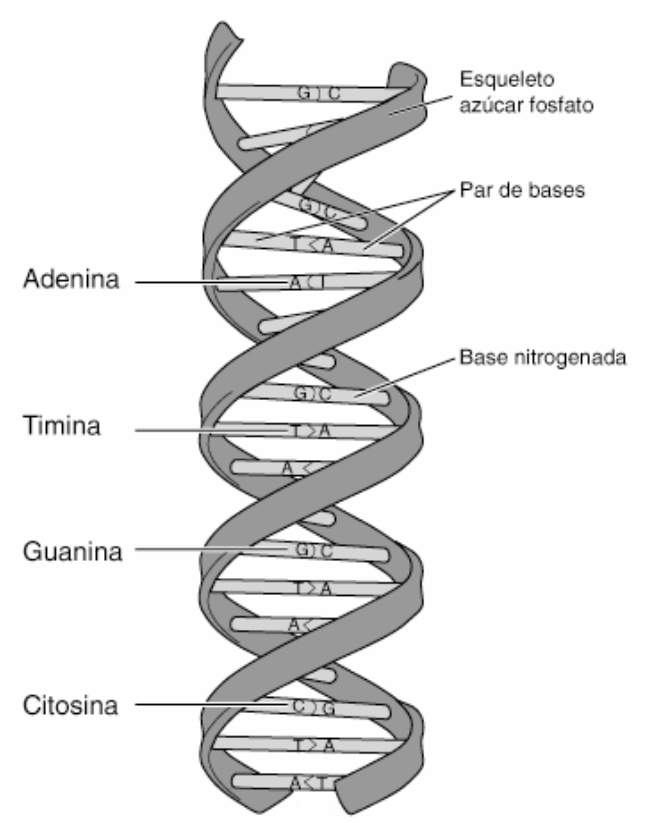

Figura 2-1. Representació d'una cadena d'ADN

Amb aquest esquema de representació es pot codificar la informació genètica **[6]**. L'alineament de seqüències es basa en que dues cadenes amb informació genètica tindran significats més semblants si s'assemblen més entre elles. Així l'alineament de seqüències consisteix en

comparar dues o més seqüències amb la intenció de representar, d'alguna manera, com de semblants són.

A les seqüències d'**ADN** hi ha la informació que es tradueix per codificar les proteïnes. Així com més s'assemblin dues cadenes d'**ADN** més podem esperar que s'assemblin les proteïnes que s'hi codifiquen.

Les diferencies entre cadenes amb mateix origen poden donar-se per tres factors. Aquests son:

- **Les mutacions:** Hi ha punts diferents entre les dues seqüències.
- **Les delecions:** Falta un tros en una seqüència respecte una altra.
- **Les insercions:** S'ha afegit un tros en una seqüència respecte una altra.

A partir d'aquests tres factors podem intentar discernir si entre dues seqüències hi ha un origen comú, per tant són homòlogues, o no. Això ho aconseguim buscant els punts comuns i provant d'arribar d'una seqüència a l'altra a partir de possibles mutacions, delecions i insercions. Quants més punts comuns i menys mutacions, delecions i insercions trobem entre elles, més semblants es poden considerar.

Una de les dificultats més grans a l'hora de comparar dues seqüències és que no te cap sentit si no és comparen alineades. Així si no trobem en quins punts s'han de comparar no es poden comparar. L'alineament consisteix en donar la millor correspondència possible entre dues seqüències a fi de poder-les comparar. Això és establir-ne els punts de major semblança i aquells on es pugui haver produït un fenomen de deleció o inserció.

El de l'alineament és un problema força complex. El primer que cal fer per resoldre'l és establir una mètrica que ens permeti comparar-ne els possibles resultats. Aquesta mètrica s'ha de basar en els diferents fenòmens biològics que poden donar-se entre seqüències.

- **Les mutacions:** Diferencies entre cadenes, es coneixen com un **miss**.
- **Les delecions:** Salts en l'alineament. Es coneixen com un **gap**.
- **Les insercions:** Donen la mateixa situació que les delecions.
- **Les coincidències:** Punts que alineem o **match**.

Donant valors positius a les coincidències (**match**) i negatius a les mutacions (**miss**) i salts (**gap**) i sumant el resultat tenim un valor per al alineament. El millor alineament és el que ens dóna el valor màxim. En aquest cas es coneix com òptim.

Un exemple amb puntuacions **+2** per al **match**, **-2** per al **miss** i **-1** per al **gap** seria:

```
G - A T E S L I K E S C H E E S E
| - | | | | - - - - - | | | | | |
G R A T E D - - - - - C H E E S E
```

Aquest seria un **alineament òptim**. Un que obté la millor puntuació possible amb la mètrica proposada:

puntuació: 10matches(+1) + 1miss*(-2) + 6gaps*(-1) = 2.*

En general es consideren dos tipus d'alineaments. Aquests són els **locals** i els **globals**. Els primers busquen donar el millor alineament possible en una zona. Els seus resultats són bons per trobar zones de màxima semblança, si comparem seqüències molt dispars, o de longituds molt diferents. Els segons busquen donar la major puntuació resultat d'alinear les seqüències completes. És adequat per comparar seqüències semblants.

Hi ha molts algoritmes que permeten obtenir alineaments a partir d'una mètrica donada. Alguns d'ells es basen en mètodes gràfics a partir de matrius de punts; d'altres en mètodes aproximats per a donar millors rendiments. Els que ens interessen per a l'estudi en aquest projecte són els mètodes que ens donen resultats òptims. Ens centrarem en dos d'ells, basats en programació dinàmica. Aquests són el **N&W**, per a alineaments **globals**, i el **S&W**, per a alineaments **locals**.

2.1.1 Algoritme Needleman&Wunsch

Es tracta d'un algoritme de **programació dinàmica**, que troba alineaments **globals òptims** entre parells de seqüències. Es basa en els mètodes gràfics que trobaven diagonals de coincidències a partir d'una matriu de punts. La **[Figura 2-2]** mostra un alineament seguint un mètode gràfic.

En aquest mètode creem una matriu on a cada punt li correspon un valor de l'alineament entre cada posició de les dues seqüències. En una segona fase podem resseguir l'alineament recorren les puntuacions obtingudes (**backpropagation**). Per a **N&W** a cada posició de la matriu li correspon el valor del millor alineament fins al moment, per a les dues cadenes.

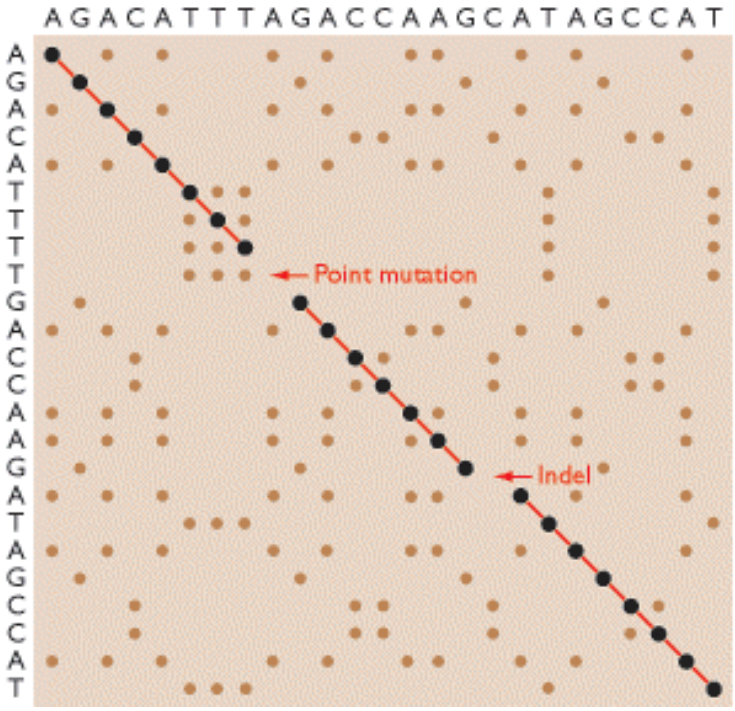

Figura 2-2. Matriu de punts i diagonals del mètode gràfic

N&W aconsegueix l'alineament en tres fases:

1. Inicialització
2. Propagació endavant
3. Propagació endarrere

Abans de començar s'han de donar valors de mètrica. Seguin una lògica biològica s'han de considerar els fenòmens de:

- **Mutacions**: No coincidència en l'alineament. **miss**.
- **Delecions**: Salts en l'alineament **gap**.
- **Insercions**: Salts en l'alineament **gap**.
- **Coincidències**: Coincidències en l'alineament **match**.

Els valors atribuïts per a cada cas s'han de correspondre proporcionalment a la probabilitat que apareguin de forma natural.

En la inicialització construïm la matriu que representa, a les files i les columnes, les cadenes a comparar. Com podem veure a la **[Figura 2-2]**, omplim la primera fila i columna amb els valors acumulats del cas en que no alineem o valor de **gap**.

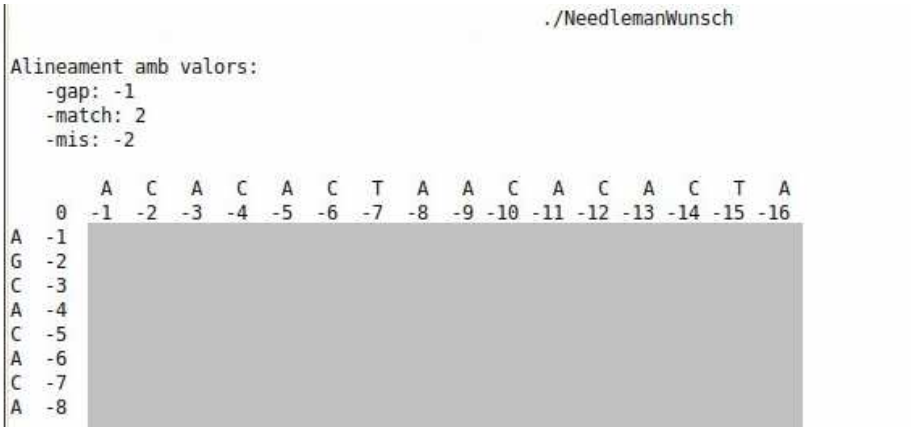

Figura 2-3. Matriu inicialitzada segons l'algoritme N&W

En la propagació endavant començant per el vèrtex superior esquerre donem a cada posició de la matriu el màxim valor entre:

- Valor de l'esquerra més penalització per **gap**.
- Valor de dalt més penalització per **gap**.
- Valor en diagonal més valor de l'alineament, segons si es un **miss** o un **match**.

Un exemple dels resultats amb valors de **gap "-1"**, **miss "-2"** i **match "2"** és el de la **[Figura 2-4]**.

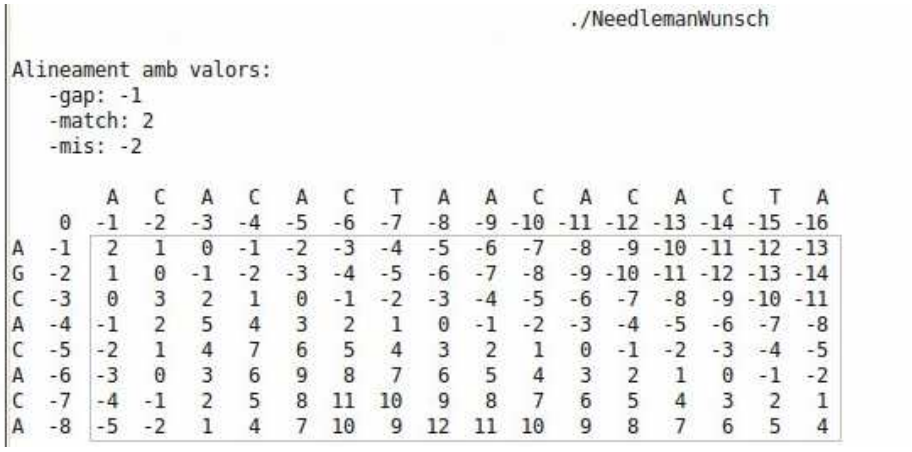

Figura 2-4. Resultat de la propagació endavant per a N&W

Per a la propagació endarrere cal tenir en compte que el valor de l'alineament és el valor de l'última posició de la matriu. A partir d'aquest, cal anar resseguint l'alineament cap amunt i a l'esquerra, seguint el camí de valors màxims com mostra la **[Figura 2-5]**.

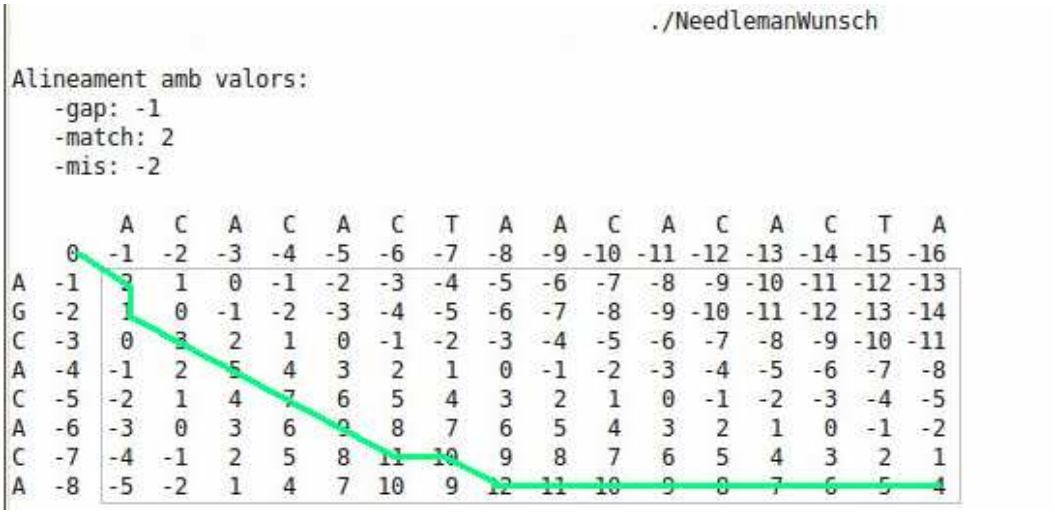

Figura 2-5. Resultat de la propagació endarrere per a N&W

El resultat de l'alineament és que les línees verticals i horitzontals es corresponen a zones amb salts en l'alineament (biològicament **delecions** o **insercions**) i les diagonals es corresponen amb les zones alineades (biològicament **mutacions**, si són diferents, o **coincidències**, si són iguals).

2.1.2 Algoritme Smith&Waterman

Es tracta d'una variant de **N&W** per a alineaments locals òptims entre parells de seqüències. Com a l'anterior construïm una matriu i l'omplim seguint tres fases. Per a obtenir el millor alineament local cal que des de tots els punts tinguem les mateixes possibilitats de ser alineats sense que ens pugui perjudicar una mala correspondència anterior. Aquesta particularitat l'aconseguim donant valor zero a tots aquells alineaments que serien negatius.

Així les tres fases per a **S&W** quedarien lleugerament modificades. Per exemple, en la inicialització la primera fila i columna quedarien inicialitzades a zero enlloc d'acumular gaps. Això permet donar la mateixa puntuació sense comptar on comencem a alinear. En la propagació endavant, seguint l'ordre de esquerre a dreta i de dalt a baix, a cada posició de la matriu li correspon el màxim entre:

- Valor de l'esquerra més penalització per **gap**.
- Valor de dalt més penalització per **gap**.
- Valor en diagonal més valor de l'alineament, segons si es un **miss** o un **mach**.
- Zero.

Els resultats els mostrem a la **[Figura 2-6]**.

```
                                                    ./SmithWaterman

Alineament amb valors:
     -gap: -1
     -match: 2
     -mis: -2

        A  C  A  C  A  C  T  A  A  C  A  C  A  C  T  A
     0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
A    0  2  1  2  1  2  1  0  2  2  1  2  1  2  1  0  2
G    0  1  1  1  1  1  1  0  1  1  1  1  1  1  1  0  1
C    0  0  3  2  3  2  3  2  1  0  3  2  3  2  3  2  1
A    0  2  2  5  4  5  4  3  4  3  0  5  4  5  4  3  4
C    0  1  4  4  7  6  7  6  5  4  5  0  7  6  7  6  5
A    0  2  3  6  6  9  8  7  8  7  6  7  0  9  8  7  8
C    0  1  4  5  8  8 11 10  9  8  9  8  9  0 11 10  9
A    0  2  3  6  7 10 10 10 12 11 10 11 10 11  0 10 12
```

Figura 2-6. Resultat de la propagació endavant per a S&W

A la propagació endarrere tenim que, per a S&W, el resultat del millor alineament no es correspon necessariament amb l'últim valor de la matriu. Aquí el millor alineament local acaba on tinguem el valor màxim. L'alineament fins a l'inici l'obtenim recorrent des del màxim, amunt i a l'esquerra, recorrent els valors més grans, fins al zero, que serà el començament de l'alineament local òptim.

No es estrany que trobem més d'un alineament amb la mateixa puntuació. En aquests casos, com el de la **[Figura 2-7]**, considerem com a alineaments locals màxims tots aquells que tinguin aquest valor. En cas d'empats durant el recorregut, la millor opció entre les possibles es considera, en alguns texts, la que dona alineament (la diagonal). Al igual que per a **N&W**, alineem les diagonals i considerem salts les verticals i horitzontals.

```
                                                    ./SmithWaterman

Alineament amb valors:
     -gap: -1
     -match: 2
     -mis: -2

        A  C  A  C  A  C  T  A  A  C  A  C  A  C  T  A
     0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
A    0  2  1  2  1  2  1  0  2  2  1  2  1  2  1  0  2
G    0  1  1  1  1  1  1  0  1  1  1  1  1  1  1  0  1
C    0  0  3  2  3  2  3  2  1  0  3  2  3  2  3  2  1
A    0  2  2  5  4  5  4  3  4  3  0  5  4  5  4  3  4
C    0  1  4  4  7  6  7  6  5  4  5  0  7  6  7  6  5
A    0  2  3  6  6  9  8  7  8  7  6  7  0  9  8  7  8
C    0  1  4  5  8  8 11 10  9  8  9  8  9  0 11 10  9
A    0  2  3  6  7 10 10 10 12 11 10 11 10 11  0 10 12
```

Figura 2-7. Resultat de la propagació endarrere per a S&W

2.2 Sobre els clústers

La paraula **clúster** ve de l'anglès i significa "ram" o manyoc. S'usa per a referir-se a molts diferents conceptes d'agrupació. En el cas que ens interessa ens hi referim com a aquell conjunt de màquines interconnectades funcionant com una unitat i treballant conjuntament en una mateixa tasca.

Aquest esquema de funcionament s'utilitza en la resolució de molts problemes de **supercomputació**, **sofware crític**, servidors i **alt rendiment**, entre d'altres. Això es per les seves possibles funcions donant redundància i escalabilitat.

L'esquema de construcció d'un clúster nomes consisteix en unir múltiples ordinadors amb una xarxa d'alta velocitat amb l'objectiu de formar un únic ordinador més potent. Amb l'esquema de la **[Figura 2-8]** aconseguim millors prestacions de forma més econòmica del que seria una única màquina amb les mateixes condicions. D'aquest esquema en podem esperar:

- Alts rendiments: **HPCC** o *High Performance Computing Clusters*
- Altes disponibilitats: **HA** o *High Availability*
- Balanceig de carga i escalabilitat: **HT** o *High Througput*

Figura 2-8. Esquema de clúster d'altes prestacions

Aquests avantatges només els aconseguirem si disposem d'algun sistema que actuï damunt els processos per administrar-ne l'execució entre els diferents nodes disponibles.

Per a propòsits científics els que acostuma a interessar és dotar-nos d'alt rendiment. En aquest cas, el que ens interessa, és obtenir gran capacitat computacional i memòria, per a dur a terme tasques molt costoses o amb moltes dades.

Una de les dificultats a l'hora de treballar amb clústers, i amb propòsits científics, és que les tasques a dur a terme siguin paral·lelitzables. Això significa poder repartir la tasca entre les màquines disponibles per fer-les treballar en paral·lel en un esquema de computació distribuïda

com el de la **[Figura 2-9]**. Cada node té la seva memòria, sense que hi hagi grans dependencies de dades entre nodes.

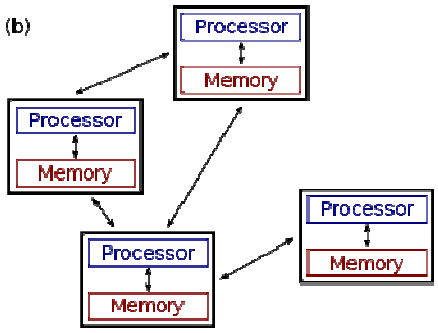

Figura 2-9. Esquema de computació distribuïda

Per a l'elaboració del present PFC disposem d'un clúster format per quatre ordinadors **multicore** connectats a través d'una **xarxa ethernet**. Cada ordenador té dos processadors i les característiques són:

- GenuineIntel(R) Pentium(R) D
- CPU a 3.4GHz
- Cache de 2048 KB de L2
- Número de cores 2
- Mida de clflush 64

Cada node disposa de 2 GB de Memoria principal i 8 GB de disc.

Tots ells funcionen amb **S.O. Linux** en la seva distribució **UBUNTU v.9** i se'ls hi ha instal·lat els paquets **BuildEssentials** i **OpenMPI**. A més per a funcionar amb **MPI** cal poder establir connexions **ssh** entre els nodes i es poden haver d'instal·lar altres paquets.

Per a l'ús de **MapReduce** en les implementacions de **Hadoop** i **Twister** veure annexes.

2.3 La paral·lelització de N&W i S&W

En una de les primeres fases de l'anàlisi dels algoritmes, en que treballem en aquest projecte, ens proposem avaluar quins càlculs es poden executar en paral·lel per a poder-ne intentar planificar una execució paral·lela **[10 i P5]**.

Per a fer aquest anàlisi tractem l'algoritme de l'alineament de **N&W** o de **S&W** en les seves diferents fases:

1. Inicialització.
2. Propagació endavant.
3. Propagació endarrere.

En la fase d'inicialització els elements de la primera fila i primera columna de **N&W** es poden calcular en paral·lel. Aquests valors tenen independència total entre ells, i només depenen de la seva posició a la matriu i de la mètrica triada (valor de **gap**). En el cas de **S&W** tots ells són zero i no cal calcular-los.

La propagació endavant la podem considerar igual per a **N&W** i **S&W**. En ella calculem els valors màxims d'alineament fins a tots els punts de cada cadena. Per al càlcul de cada punt prenem les dades dels alineaments fins al moment. D'aquesta manera no podem calcular cap valor de la matriu sense haver calculat amb anterioritat els seus valors superior, esquerra i superior-esquerra, tal i com veiem a la **[Figura 2-10]**.

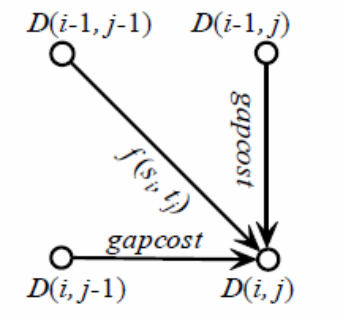

Figura 2-10. Esquema del sentit de les dependencies a Smith&Waterman i Needleman&Wunsch

A partir de aquestes dependencies de dades podem plantejar-nos la propagació endavant com a una sèrie de cicles amb una sèrie de càlculs que es poden fer en cada un. Com mostra la **[Figura 2-11]** en una primera iteració només podríem calcular el valor de la posició (2,2), en la segona en paral·lel (3,2) i (2,3), en una tercera (4,2), (3,3) i (2,4), creixent en un esquema de propagació de dependències en onada o gota d'aigua.

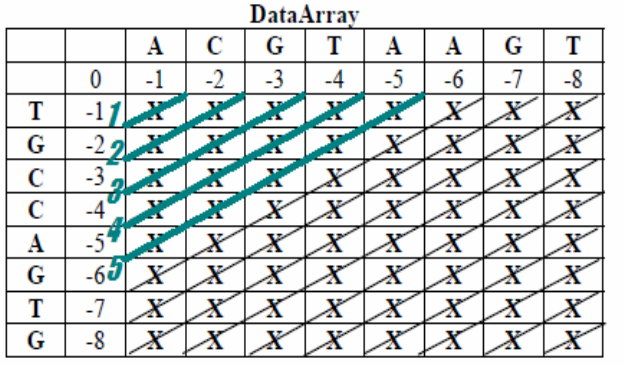

Figura 2-11. Esquema d'execucions paral·leles per cicles

La propagació de dependencies creix fins que arribem a la longitud de la més petita de les cadenes a comparar, es manté fins a la mida de la cadena més gran i després es va reduint fins al càlcul de l'últim valor, que és el resultat de l'alineament per a **N&W**. En el cas de **S&W** durant el càlcul cal guardar els punts de màxim valor de l'alineament per no haver de cercar-los després.

A la propagació endarrere s'ha de recórrer pas per pas, seguint el camí de màxims i no ens podem permetre de fer-ho en paral·lel.

2.4 El paradigma MapReduce

MapReduce [5 i P6] és un mètode introduït per **Google** per donar suport a la computació paral·lela sobre grans col·leccions de dades en grups de computadors. Esta inspirat en la programació funcional i concretament en les operacions de **map** i **reduce**. En les operacions **map** les dades s'agrupen en parells clau/valor, en el que es diu dades intermèdies, i en les operacions **reduce** s'operen per obtenir el resultat final.

La manera en com opera **MapReduce** és la de usar grans quantitats de maquines (nodes) organitzats en estructura de **Master-Worker [Figura 2-12]**. En una fase de **map** el **Master** distribueix subproblemes entre els **Workers**. Aquests operen en paral·lel per crear les dades intermèdies. El resultat de la fase de **map** és l'entrada per a la fase de **reduce**. En aquesta un cert nombre de **Workers** les operen per a obtenir el resultat final.

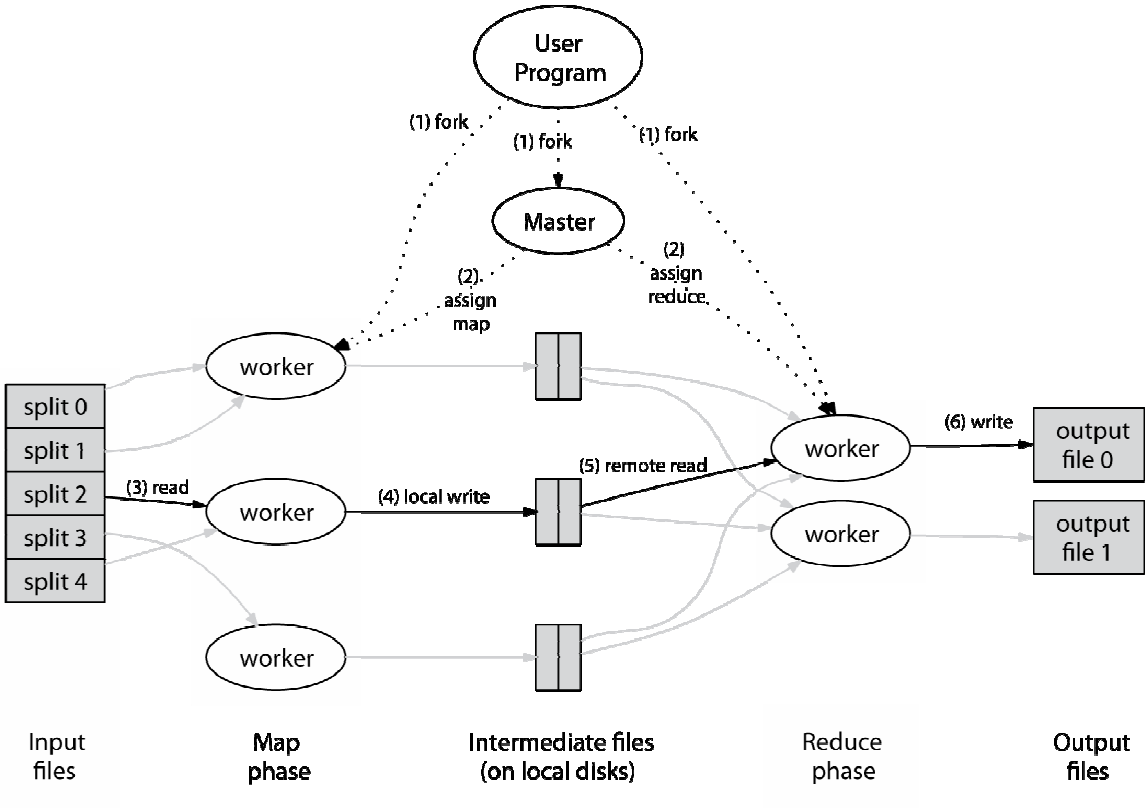

Figura 2-12. Esquema de funcionament de MapReduce

El nombre de **Workers** que operen en cada una de les fases és un paràmetre crític de configuració. També ho són les mides de les particions en subproblemes de la tasca inicial i característiques dels nodes com les mides dels **buffers**, etc.

En un esquema de **MapReduce** ha d'haver-hi independència de dades entre els **workers** per poder operar-les totes alhora. Un exemple tipus per a veure el funcionament d'aquest paradigma és el del comptador de paraules, conegut com a **Wordcount**.

2.4.1 Exemple de MapReduce: Wordcount

En aquest exemple s'usen les primitives **map** i **reduce** per a comptar l'aparició de cada paraula en un conjunt de documents.

El codi consistiria en per a cada fitxer d'entrada un **map** crea per a cada paraula del text una dada intermèdia de tipus classe/valor amb la paraula i el valor "1". Aquestes dades són les entrades del **reduce** que per a cada paraula retornarà el nombre d'aparicions.

```
void map(String name, String document):
  // name: document name
  // document: document contents
  for each word w in document:
    EmitIntermediate(w, "1");

 void reduce(String word, Iterator partialCounts):
  // word: a word
  // partialCounts: a list of aggregated partial counts
  int result = 0;
  for each pc in partialCounts:
    result += ParseInt(pc);
  Emit(AsString(result));
```

Figura 2-13. Codi MapReduce per al Wordcount

La **[Figura 2-13]** representa un codi d'exemple de la funcionalitat dels operadors **map** i **reduce**.

2.4.2 Framework de Hadoop

Existeixen moltes implementacions de **MapReduce**. La més important entre les **Open Source** és la del projecte **Hadoop** d'**Apache Software Foundation [2]**. És un **framework** per suportar aplicacions distribuïdes per al treball intensiu de dades.

Les avantatges d'usar un **framework** com el presentat rauen en que aquests inclouen una sèrie d'optimitzacions. Exemples d'aquestes en són el control dels accessos als sistemes de fitxers, que optimitzen el pas de dades entre els nodes dels clústers, la tolerància a fallades i característiques sobre la seguretat.

Aquest **framework** en concret incorpora el seu propi sistema de fitxers pensat per a grans fitxers, en múltiples màquines, el **HDFS** (**Hadoop Distributed File System**). Aquest sistema organitza les dades entre els nodes amb redundància com es mostra a la **[Figura 2-14]**.

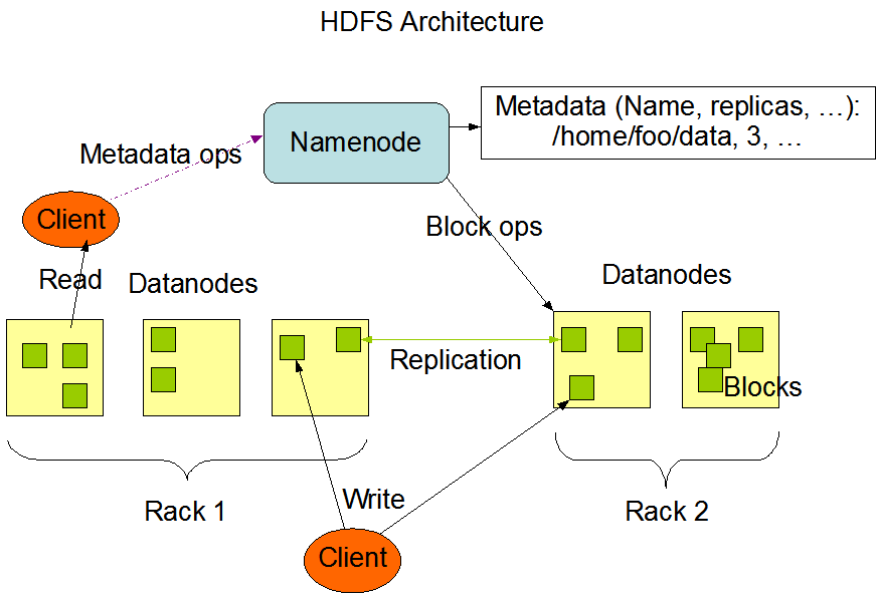

Figura 2-14. Esquema de funcionament de l'arquitectura HDFS

2.5 Introducció a MapReduce Iteratiu

Entre les limitacions més evidents de **MapReduce** hi ha la de que no permet operar amb entrades entre les que hi ha dependències de resultats o en aquells problemes en que aquests resultats es calculen de forma iterativa i on els d'una iteració serveixen d'entrada a la següent o com a condició de finalització.

Així **MapReduce** és una manera d'organitzar el càlcul paral·lel si aquest pot donar-se sense condicions i dependències, aquestes limitacions fan més adequat **MapReduce** per a uns problemes i menys d'altres.

Per al nostre problema calia introduir la iterativitat o el problema no es podia resoldre d'una manera avantatjosa. Per això varem buscar quines opcions hi havia. Entre les analitzades destaquem:

- **Pregel** [**P1**]: Infraestructura de **Google Research** per al procés de dades en grafs de llargues escales.
- **Twister** [**12**]: Versió de **MapReduce** que permet planificar les tasques seguint un model Iteratiu. És un projecte de la Universitat d'Indiana.
- **Dryad Linq** [**P1**, **8**, **9**]: Projecte de Microsoft per a la resolució de problemes de gran escala com una plataforma per a la computació distribuïda.
- **CGL Hadoop** [**P1**]: Implementació de **MapReduce** basada en streaming que inclou suport per a computar problemes Iteratius.

Totes haurien de poder resoldre problemes com el que tenim, però alhora totes presenten algun problema. La alternativa de **Pregel** obliga a fer la difícil conversió del algoritme iteratiu a un problema de recorre grafs sense mida fixa. La opció de **Dryad Linq** l'hem de descartar perquè el sofware de desenvolupament no està encara disponible i només podríem intentar resoldre el problema teòricament. La possibilitat de **CGL** sembla que es solapa amb la de **Twister**, dons no es troba pràcticament informació. Així nomes ens queda per estudiar com funciona **Twister MapReduce**.

Per a entendre com funciona **Twister** ens centrarem en dels problemes tipus que permet resoldre. Aquest és l'algoritme **Kmeans** per a classificació.

2.5.1 Twister MapReduce i Kmeans

Anem a repassar com funciona **kmeans**:

A partir d'un grup d'observacions **(x_1, x_2, ..., x_n)**, un vector de nombres reals de **d** dimensions, l'algoritme de kmeans per a la classificació permet repartir les **n** observacions entre **k** grups **(k<n) S={S_1, S_2, ..., S_n}** tals que es minimitzi la suma de les distàncies als centres dels grups segons la fórmula:

$$\arg\min_{\mathbf{S}} \sum_{i=1}^{k} \sum_{\mathbf{x}_j \in S_i} \|\mathbf{x}_j - \boldsymbol{\mu}_i\|^2$$

on μ_i és la mitjana dels punts de S_i.

Minimitzar les distàncies s'aconsegueix a partir de recalcular els centres dels grups de forma iterativa, repartint els punts entre els nous centres per proximitat i recalculant l'error fins obtenir un error inferior a un d'objectiu.

Això podem fer-ho aprofitant les prestacions de **MapReduce** usant **Twister** seguint l'esquema de la **[Figura 2-15]**.

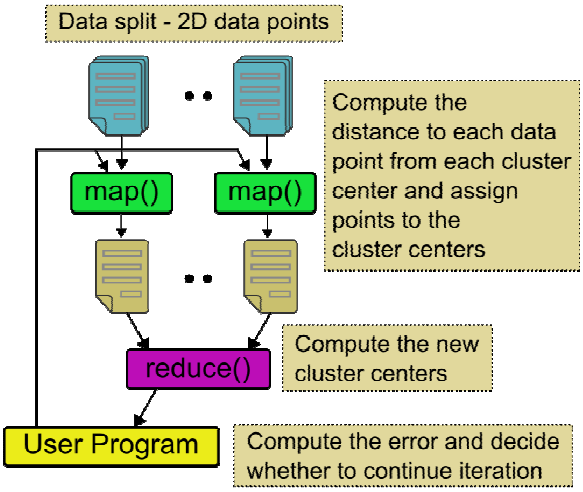

Figura 2-15. Esquema per a la resolució de Kmeans amb Twister MapReduce

En l'esquema s'observa la diferència respecte la versió original. Per a resoldre **Kmeans** les dades d'entrada són els centres dels grups. En un cicle de **MapReduce** recalculem les distàncies. El programa decideix segons l'error obtingut per als nous centres si seguim iterant. En cas de seguir iterant les dades obtingudes en el cicle anterior seran les dades d'entrada per a la següent iteració.

Per a fer possible aquestes prestacions **Twister** introdueix noves classes a les implementacions de **MapReduce** amb mètodes per capturar els resultats per poder usar-los després. Un exemple d'aquests és mostra a la **[Figura 2-16]** amb el mètodes **getResults()**.

```
...
TwisterModel driver = new TwisterDriver(jobConf);
Driver.configureMaps(partitionFile);
...
While(!complete){
monitor=driver.runMapReduceBCast(cData);
Monitor.monitorTillCompletion();
DoubleVectorData newCData=((KMeansComviner)
   driver.getCurrentCombiner()).getResults();
...
}
```

Figura 2-16. Exemple de codi de Twister MapReduce

Observem algunes limitacions per a la planificació de problemes iteratius de mides no fixes, com en el nostre cas, que la mida del càlcul paral·lel creix en forma de gota que propaga una onada.

Capitol 3 Fases de disseny i implementació

El disseny de les diferents versions dels codis està motivat per les limitacions de l'anterior i prova de millorar-ne el rendiment.. En totes les fases s'ha tingut en compte que l'objectiu és millorar les prestacions en temps i memòria per a quan el problema es fa gran.

En les solucions no tenim en compte la fase de propagació endarrere, o **backpropagation**, per no ser interessant per al estudi, ja que s'ha de fer de manera seqüencial necessariament.

Les explicacions són equivalents, en general, per als problemes amb **N&W** i **S&W**.

3.1 La programació sèrie

La proposta de solució aquí fa tots els càlculs de forma seqüencial. El problema principal està en com plantegem la solució tenint en compte que no coneixem la mida de les entrades i que aquesta pot ser molt gran. Si la mida no ens importés una solució senzilla seria tal i com es mostra a la **[Figura 3-1]** en pseudocodi.

```
read A                                    (només a N&W)
read B                                    (només a S&W)
for i=0 to length(A)-1                    S(A(i),B(j)): resultat comparació
    F(i,0) ← gap*I                        gap: valor del salt
    F(i,0) ← 0
  for j=0 to length(B)-1
    F(0,j) ← gap*j
    F(0,j) ← 0
  for i=1 to length(A)
    for j = 1 to length(B)
    {
      Choice1 ← F(i-1,j-1) + S(A(i-1), B(j-1))
      Choice2 ← F(i-1, j) + gap
      Choice3 ← F(i, j-1) + gap
      Choice3 ← 0
      F(i,j) ← max(Choice1, Choice2, Choice3, Choice4)
    }
```

Figura 3-1. Pseudocodi del càlcul de la matriu F de resultats a S&W i N&W

Això obliga a tenir estructures **A** i **B** tant grans com grans siguin les entrades, i **F** una matriu de resultats, de la propagació endavant, tant gran com el producte entre les mides de les entrades. Això pot ser tant gran que em de mirar d'aplicar estratègies de **Divide&Conquer [P4]**.

Per fer-nos una idea de la magnitud de **F** posem uns exemples. Amb entrades **A** i **B** de 1000 caràcters, si els valors són enters (4bytes) la mida que ocuparia **F** al disc es calcularia:

1.000 * 1.000 * 4 = 4.000.000 bytes Aproximadament 4 MB.

Si el problema creix tenim:

A:	10.000 bases	**B:**	10.000 bases	**F:**	aproximadament 400 MB
A:	100.000bases	**B:**	100.000bases	**F:**	aproximadament 40 GB
A:	1 milió bases	**B:**	1 milió bases	**F:**	aproximadament 4.000 GB

Així la mida en memòria de les dades que creem en l'aplicació dels algoritmes implementats creix en ordres de **4*(mida^2).** Com que aquests resultats s'han de mantenir per a poder fer la propagació endarrere em de pensar una solució.

Per fer tractables aquestes magnituds de dades la solució en aquest apartat consistirà en dividir el problema en blocs de mides fixes i anar guardant el resultat de cada bloc en un fitxer.

3.1.1 Esquema del disseny

En termes generals la proposta divideix el problema en problemes més petits. La idea es pot veure a la **[Figura 3-2]**.

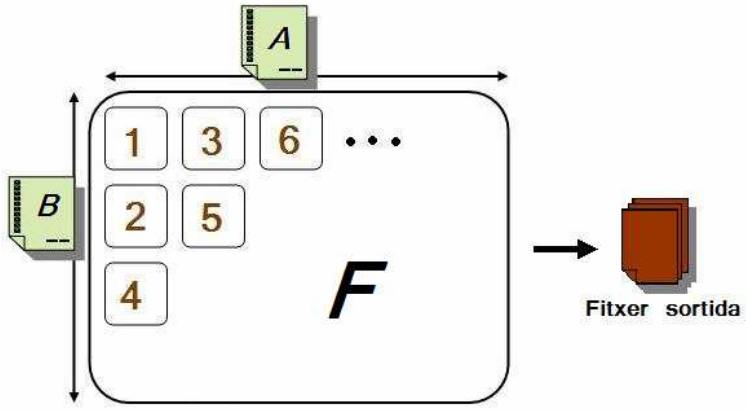

Figura 3-2. Esquema de la solució seqüencial de S&W i N&W

Segons aquesta proposta calculem parts del càlcul de manera iterativa. Primer calcularem els alineaments del bloc corresponent a la primera part de **A** i la primera part de **B**, el **bloc 1**. Un cop tenim els resultats calculem el **bloc 2**, en acabat el **3** i així fins acabar amb tots els blocs en que partim el problema. S'ha de tenir en compte que els resultats de cada bloc s'usen per als següents seguint l'ordre de dependència. Així en blocs la dependència es converteix, com representa la **[Figura 3-3]**, en que per a cada bloc de la perifèria, necessitem els valors de la inicialització per calcular-ne els alineaments, i cada bloc a l'interior necessita la última fila del de dalt i la última columna del de la esquerra.

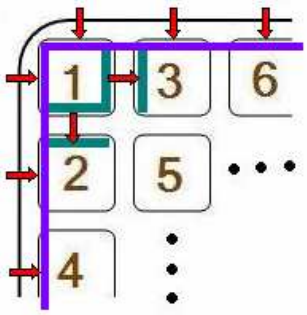

Figura 3-3. Dependències a la solució per blocs

La mida dels blocs és un paràmetre que pot variar. L'anàlisi provarà de respondre a si hi ha valors més bons que altres. Una funció de càlcul opera cada bloc i guarda el resultat en un únic fitxer. Aquesta funció calcularà la matriu d'alineaments de la seva porció de treball iterativament tal hi com es fa en el pseudocodi. Aleshores cada bloc escriu els seus resultats a la posició adequada del fitxer, successivament com la representació de la **[Figura 3-4]**, segons la seva posició respecta la matriu imaginària **F**.

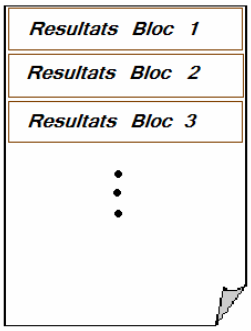

Figura 3-4. Organització del fitxer de sortida

Per mantenir les dades necessàries de les files i columnes, fins a que s'operin els blocs que les necessiten, ens cal una estructura que pugui mantenir tots els valors de les últimes files i columnes de la diagonal de blocs actual, que s'usaran a la diagonal de blocs següent. A la

[Figura 3-5] representem en colors les dades a ser guardades a cada iteració. D'això en direm propagació de dades en forma d'onada.

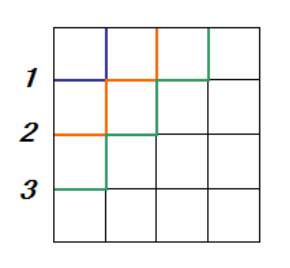

Figura 3-5. Propagació de la onada de dependències per a les tres primeres diagonals

La mida de l'estructura haurà de mantenir tantes files i columnes de dades a propagar com gran pugui arribar a ser la diagonal. Això serà tant gran com el número de blocs en que es puguin partir les dades del fitxer més curt.

3.1.2 Codi

En aquest apartat en proposem mostrar el pseudocodi i comentar parts del codi implementat que considerem poden ser interessants. No pretenem posar codi ni comentar aquelles parts que són de correspondència directa amb els algoritmes. Sí que creiem que cal incloure comentaris sobre aquelles parts que tenen a veure amb la organització de la execució i la gestió de les dades.

El pseudocodi de la **[Figura 3-6]** seria el que organitza el càlcul dels blocs en que partim el problema.

```
BlocsA = Calcular numero de blocs per A()
BlocsB = Calcular número de blocs per B()

InicialitzarEstructures()

NumeroDiagonals = blocsA + BlocsB – 1
NumeroBlocsDiagonal = 1
FilaInicial = 1
ColumnaInicial = 1
Obrir fitxer sortida()

for i = 0 to NumeroDiagonals
{
  Fila = FilaInicial
  Columna = ColumnaInicial
  for i = 0 to NumeroBlocsDiagonal
```

```
    {

      CalculaBloc(Fila,Columna)

      Fila--
      Columna++
    }
    Si(i>BlocsA && i>BlocsB)
      NumeroBlocsDiagonal--
    Sino Si (i<BlocsA && i<BlocsB)
      NumeroBlocsDiagonal++
    Si (i >= BlocsB)
      ColumnaInicial++
    Sino
      FilaInicial++
  }

  Tancar fitxer sortida()
```

Figura 3-6. Pseudocodi de la execució per blocs

Necessitem banderes en que comptem en quina diagonal estem i de les coordenades **fila** i **columna** del bloc que calculem. Seguint el pseudocodi veiem que el càlcul es fa diagonal per diagonal i cada diagonal de baix a l'esquerra a dalt a la dreta.

La funció **InicialitzarEstructures()** crearia dues estructures de les mides de la màxima diagonal. En una guardaríem les dades que anéssim llegint del fitxer per no repetir les lectures a cada iteració. En l'altre guardaríem les dades de la onada de dependència entre diagonals.

La funció **CalculaBloc()** seria la encarregada d'aplicar l'algoritme però necessita les dades amb les que fer-ho. Així **CalculaBloc()** necessita la fila i columna del bloc del que ha de calcular:la matriu d'alineament. També ha de tenir accés a les estructures, que han de ser globals, i al punter del fitxer de sortida. Com que els blocs no sempre son iguals (això passa quan algun dels fitxers s'acaba) també ha d'haver dues variables per saber les mides de cada bloc. Aquesta funció crea una matriu de la mida de les dades a cada bloc, aplica l'algoritme d'alineament sobre elles i en guarda les dades en el fitxer en la posició que li correspon. Fet això ha de guardar la nova fila i columna de la onada a propagar sense sobreescriure dades necessàries per al següent bloc (calen dues estructures per fer-ho).

Per poder declarar estructures prou grans em d'usar memòria dinàmica. Això ho fem per a la declaració de les estructures i de la matriu del càlcul de cada bloc. Per a les estructures dels fitxers declarem estructures de les mides que ens interessen de punters. Llavors ja demanarem memòria perquè cada punter apunti a les dades llegides del fitxer com es fa a la **[Figura 3-7]**.

```
char* punter_fitxerA[BlocsA];
char* punter_fitxerB[BlocsB];
for i = 0 to BlocsA
  punter_fitxerA[i] = NULL;
for i = 0 to BlocsB
  punter_fitxerB[i] = NULL;

…

punter_fitxerA[Fila] = new char[MidaBloc];

…

punter_fitxerB[Columna] = new char[MidaBloc];
```

Figura 3-7. Pseudocodi de l'ús de memòria dinàmica per les lectures dels fitxers d'entrada

Així cada cop que necessitem una posició del fitxer, si aquesta apunta a **NULL** la llegim i guardem en memòria nova i sinó la usem sense llegir de fitxer.

De manera equivalent mantenim les estructures de la onada de dependència de últimes files i columnes amb la diferencia que aquestes s'han d'inicialitzar a **zeros** per a **S&W** i al valor de **posició*gap** a **N&W**.

La matriu de càlcul la declarem, com es fa al codi de la **[Figura 3-8]**, com un vector de punters a vectors d'enters perquè sinó tindríem molt limitada la mida permesa per al compilador.

```
int* F[MidaBloc]
for i=0 to MidaBloc
{
  F[i]=new int[MidaBloc]
}
```

Figura 3-8. Pseudocodi de la declaració de la matriu de càlcul

Hem de tenir la precaució d'alliberar aquesta memòria perquè sinó podem acabar amb els recursos, si el problema és prou gran.

3.1.3 Anàlisi

En aquesta fase el en interessa analitzar és:

- A què es dedica el temps amb la solució proposada.
- Com influencia la mida dels Blocs a aquests temps.
- Quantitat de memòria que utilitzem.

També és el moment d'entendre millor el problema i de valorar de quina manera podem planificar una execució més eficient. De l'anàlisi volem treure'n una idea clara de quin percentatge de temps dediquem a cada etapa en la execució i com varien els temps segons paràmetres com la mida de les entrades o dels blocs amb que fem la partició de la tasca a fer.

També és interessant analitzar si l'ús de memòria és una característica limitant. Això ho podem veure en part per la influencia que hi té la mida dels blocs ja que no mantenim totes les estructures en memòria sinó només les del bloc en execució.

3.2 L'ús de Threads i memòria compartida

En aquesta fase volem mirar de solapar tot el càlcul possible dins la mateixa màquina i aprofitar que disposem de màquines **multicore**. En la fase anterior hem vist com a cada bloc s'utilitza el temps. Hi ha un temps de càlcul, un temps de inicialització de dades i un temps en el que s'escriu al fitxer. Usant **threads [11]** el que es volem aconseguir és tenir, sempre que sigui possible i en el grau que sigui possible, la màquina aprofitada.

Definim les fases d'execució en tres parts:

1. Inicialització
2. Càlcul
3. Accés a memòria

Si podem mantenir diferents processos amb parts de la tasca a executar, podem fer que mentre un accedeix a memòria, un altre s'inicialitzi o calculi. Si a més disposem d'equips amb més d'una **CPU** això es pot fer amb més paral·lelisme encara. D'aquesta manera s'aprofiten els temps en que les tasques esperen dades de memòria o per anar a memòria. Aquests temps són els

que s'anomenen estat bloquejat en els diagrames d'execució de processos com el de la **[Figura 3-9]**. Baris processos permeten ocupar els diferents estats i no desaprofitar capacitat de **CPU**.

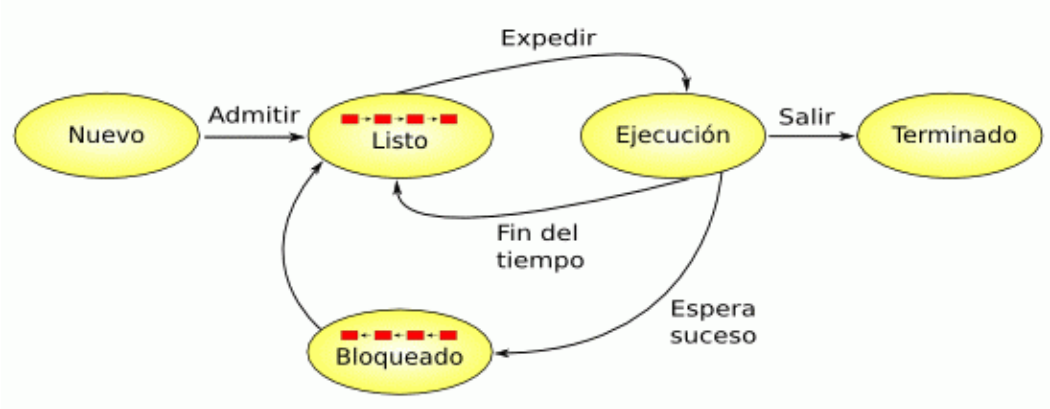

Figura 3-9. Estats d'un procés i transicions

En un esquema ideal, si les operacions amb **CPU** i les d'accés a memòria s'executessin amb el mateix temps amb aquest esquema podríem obtenir molts guanys.

La proposta utilitza la mateixa idea que l'anterior aprofitant la possibilitat de calcular tots els blocs d'una mateixa diagonal en paral·lel, seguint l'ordre de dependències en forma d'onada.

3.2.1 Esquema del disseny

L'esquema per a aquesta proposta parteix de l'anterior i hi inclou l'ús de **threads** per fer tots els blocs possibles en paral·lel. Ara podem distingir diferents processos per a cada bloc de cada diagonal i un procés principal.

El procés principal llegeix les mides del fitxer, prepara les dades per a fer les execucions i crea un **thread** que fa el càlcul de les submatrius i guarda els resultats en un únic fitxer. La diferència amb l'anterior és que ara no esperem els resultats de cada bloc per iniciar el següent, sinó que s'inicien tots els que es poden calcular en l'ordre de les dependencies, a cada cicle.

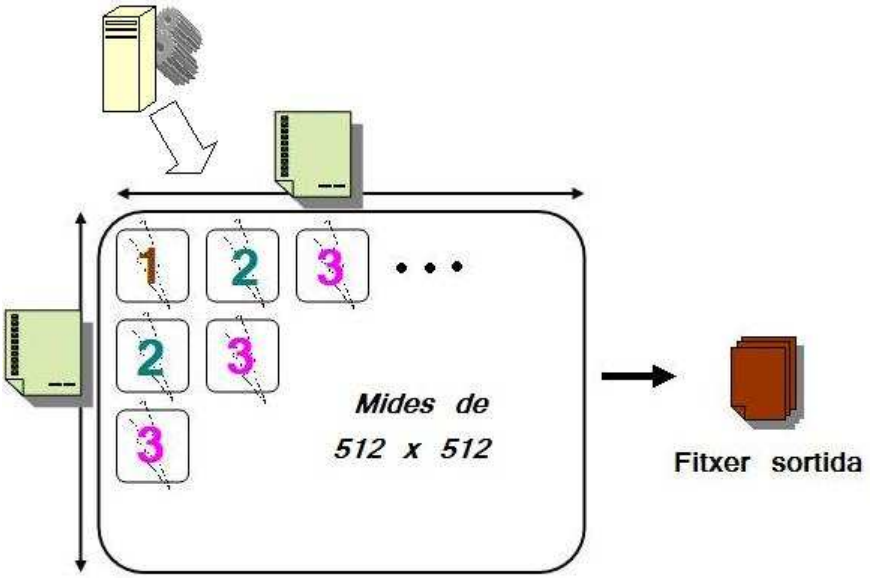

Figura 3-10. Esquema de la solució usant Threads de S&W i N&W

A la **[figura 3-10]** veiem com tots els blocs poden ser calculats en el mateix cicle. Aleshores varis processos poden ser executen concurrentment, solapant el càlcul amb els accessos a memòria. El tractament de les dades i del fitxer amb els resultats es fa exactament igual que a l'esquema anterior.

3.2.2 Codi

En aquest apartat és interessant comentar del codi aquelles coses que difereixen de l'anterior. La diferencia més notable és la de usar un **thread** per a fer l'execució dels blocs.

Com que usarem un nombre variable de **threads** declarem un vector d'aquests amb el nombre en que voldrem utilitzar-los com a la **[Figura 3-11]**. El nombre màxim serà la mida de la diagonal més gran.

```
pthread_t* thread = new pthread_t[num];
```

Figura 3-11. Declaració d'un vector de threads

Cada un d'aquest executarà una funció que tindrà com entrada una estructura amb totes les dades necessàries per fer la seva porció de càlcul. Veiem a la **[Figura 3-12]** l'esquema de l'estructura i la crida a la funció que s'executarà en un procés propi.

```
struct data_threads{
        int m_iNum_fil;
        int m_iNum_col;
        int m_iEscSize;
        int m_iDalSize;
        int* m_pEsc;
        int* m_pDal;
        char* m_cDataEsc;
        char* m_cDataDal;
};

...

pthread_create( &thread[i], NULL,thread_tasc, (void*)&data1[i]);
```

Figura 3-12. Estructura de dades i crida a crear un thread

Caldran vectors d'estructures de la mida de les diagonals de blocs i les haurem de tenir per duplicat per poder guardar els resultats d'una per a usar com a entrades a l'anterior. Tota aquesta memòria la reservem i alliberem usant memòria dinàmica.

El càlcul a cada **thread** serà exactament igual que per al cas anterior. Tota la inicialització la fa el procés principal ja que així els fitxers només s'accedeixen un cop i ens estalviem problemes de control d'accés a les variables que s'haurien de compartir. Les diferents parts de l'estructura son:

1. **Inicialització**: La fa el procés principal
2. **Càlcul**: El fan els diferents processos
3. **Guardar resultats**: Ho fan els diferents processos

El cicle d'execució consistirà en:

1. Iniciar i executar cada un dels blocs que es puguin calcular d'una diagonal
2. Calcular en processos paral·lels dels blocs
3. **Join** de cada procés
4. Copia de les files i columnes finals de cada bloc

3.2.3 Anàlisi

Com en l'anterior versió aquí també volem analitzar els temps que triguen les diferents tasques i com varien amb la mida de les estructures, i segons les mides dels fitxers. Ara però també voldrem comparar els resultats amb l'execució de la versió anterior per veure com és la millora.

En aquest esquema la importància de la mida de les estructures és encara més important perquè ara interessa que el temps de càlcul i el temps d'accedir a la memòria siguin el més semblant possibles ja que el més gran d'aquests serà el limitant a la hora de la paral·lelització del procés.

En quant a la memòria usada tenim en compte que ara ja no necessitem només per a un bloc sinó per a tants com processos iniciem en paral·lel.

Les diferents execucions a provar són:

- Mesures de temps segons número de processos en execució en paral·lel
- Mesures de temps variant la mida dels blocs
- Càlculs d'utilització de memòria
- Comparacions de rendiment

També és un objectiu de l'anàlisi valorar els punts febles i els límits en les millores de rendiment que podríem arribar a assolir.

3.3 Programació paral·lela, MPI i la memòria distribuïda

Un cop vistos els resultats en els esquemes anteriors volem proposar una solució per a que el càlcul es faci conjuntament entre una xarxa d'ordinadors comunicats entre ells amb les primitives de **Message Passing Interface** (**MPI**) **[3,4 i P3]**. La idea és distribuir les tasques que puguin fer-se de forma paral·lela entre un cert nombre de nodes que n'efectuarien el càlcul i en guardarien els resultats en fitxers. Per a fer això partirem d'una idea basada en politiques **Master-Worker**, on el **Master** gestiona l'execució i envia missatges als **Worker** per activar-los, i aquests responen quan han acabat.

Plantejar una solució paral·lela distribuïda comporta tenir en compte tota una sèrie de variables i condicions. El control de l'execució, la sincronització, els accessos a les dades i l'ús de memòria distribuïda, en lloc de memòria compartida en són els més significatius.

El control de la execució:

Es fa a partir d'un sistema **Master-Worker** on el **master** inicia els **workers** i aquests comuniquen la fi i envien resultats.

Accessos a les dades:

S'ha de tenir en compte que hi ha moltes dades que són comuns als nodes i d'altres que s'han de sobreescriure o reutilitzar. Cal molt control per a assegurar que les dades que es llegeixin o escriguin siguin usades correctament.

Ús de memòria distribuïda:

Cada node usa les seves pròpies dades. Llegeix del node **Master** el fitxer i copia tot el que necessita. Si necessita altres dades s'han d'enviar i rebre usant de funcions **MPI**.

3.3.1 Esquema del disseny

A partir de la anterior implementació volem millorar els resultats. Ara disposem d'una xarxa interconnectada de màquines que poden treballar independents les unes de les altres.

En l'esquema de funcionament un node executa el programa que distribueix les tasques i els altres esperen missatges del principal, per fer les parts de còmput que se'ls demani. El node principal inicialment calcula la mida dels fitxers per saber en quantes parts dividirà la tasca, o quantes tasques s'han de passar als altres nodes. D'aquestes divisions en direm en endavant **macroblocs**. Cada **macrobloc** estarà format per un nombre enter de blocs, com els de la versió anterior, i serà la mida de la tasca que es demana als **workers**. Veiem aquesta divisió a la **[Figura 3-13]**.

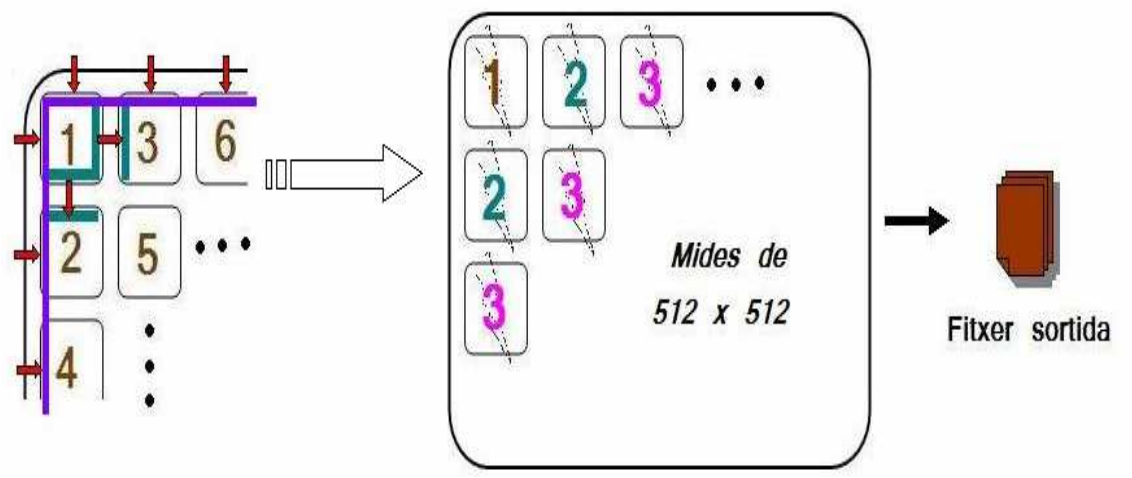

Figura 3-13. Esquema de la divisió en macroblocs a MPI

Per a poder fer el seu càlcul, cada node necessita les dades inicials del **macrobloc** per salvar les dependències. Seran les dades inicials que no podem calcular nosaltres directament si no som el primer **macrobloc**. Pel que fa a les dades dels fitxers aquestes es llegeixen un cop des de cada node quan les necessita.

Dins cada node es fan els càlculs de l'alineament de la seva zona tal i com es feia en la versió de **threads**, dividint els **macroblocs** en blocs i creant un fitxer de resultats únic per a cada tasca.

L'esquema general de tot el procés és el de la **[Figura 3-14]**. Mostra com, tant el node principal com els **workers**, tenen accés als fitxers d'entrada, com el node principal distribueix la tasca en **macroblocs**, com aquests s'executen segons la versió de **threads,** dividint la tasca en blocs, i com cada node crea els seus fitxers de sortida amb les dades de l'alineament de cada zona.

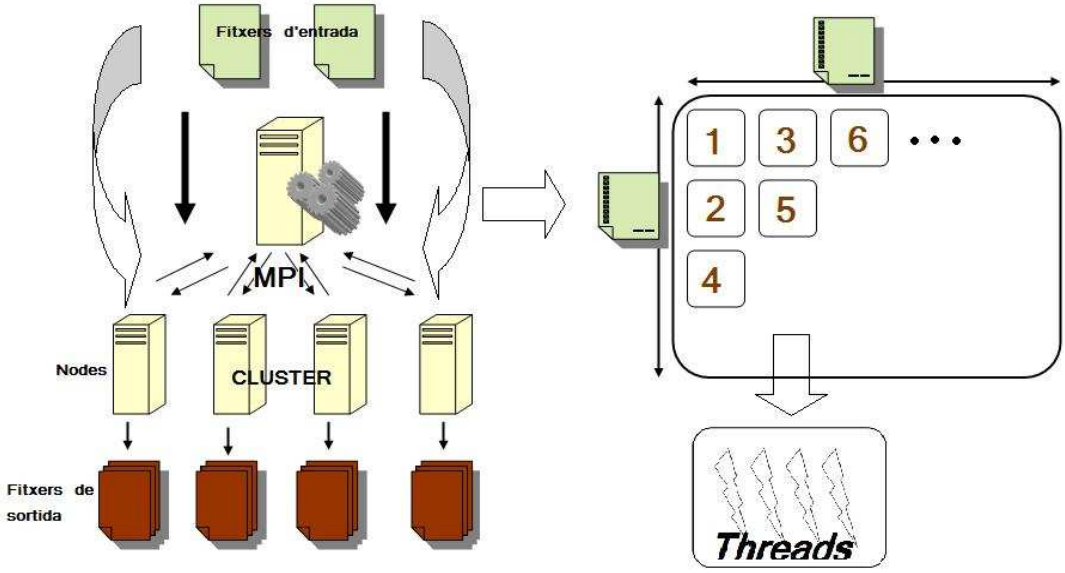

Figura 3-14. Esquema de la solució usant MPI de S&W i N&W

La solució **MPI** té com a paràmetres de configuració el nombre de nodes a executar, la mida dels blocs i el nombre de blocs que volem per cada **macroblocs**.

3.3.2 Codi

En aquesta versió ja hem dit que s'utilitza el codi de la versió amb **threads** per fer tots els càlculs per a cada **macrobloc**. Així el que interessa comentar és com funciona MPI, com es fa la gestió dels nodes i de les dades i quines comunicacions s'utilitzem per a la sincronització entre el node principal i els **workers**.

A la **[Figura 3-15]** veiem com s'inicialitzen els varis nodes després de fer una crida a execució MPI.

```
/***** Initializem tots els nodes *****/
MPI_Init(&argc, &argv);
/***** Preguntem numero de nodes *****/
MPI_Comm_size(MPI_COMM_WORLD, &num_nodes);
/***** Ens identifiquem el numero de node *****/
MPI_Comm_rank(MPI_COMM_WORLD,&taskid);
printf ("MPI proces %d iniciat...\n", taskid);
```

Figura 3-15. Crides a inicialitzacions MPI

Les crides **MPI_Comm_size** i **MPI_Comm_rank** retornen el número de nodes i l'identificador del node respectivament. Usem l'identificador per a referir-nos als diferents nodes en les comunicacions. L'identificador també serveix per saber quina és la nostra tasca. Així, com veiem a la **[Figura 3-16]**, els nodes faran la tasca de **master** o **worker** segons el número d'identificador.

```
if (taskid == 0){
        // Tasca del master
        ...
}

if (taskid != 0){
        // Tasca dels workers
        ...
}
```

Figura 3-16. Mode d'identificació de tasques als nodes

Un cop tenim el nombre de **macroblocs** a executar toca repartir-los entre els nodes segons si ja es poden calcular segons les dependències. A la **[Figura 3-17]** mostrem el codi amb que hem fet aquest control. Cada envio demana a un node dels disponibles el càlcul d'un **macrobloc** i rebem la última fila i columna d'aquell **macrobloc**.

```
int mida_diagonal=1;
int sends_pendents=1;
int reciv_pendents=0;
int iteracio=1;
int fila_inicial=1;
int columna_inicial=1;
int fila=1;
```

```
int columna=1;

while (mida_diagonal>0){

        fila=fila_inicial;
        columna=columna_inicial;
        sends_pendents=mida_diagonal;
        reciv_pendents=0;
        while(sends_pendents>0){
                for (dest=1; dest<num_nodes; dest++) {
                        if(sends_pendents>0){
                                //enviem les peticions
                                fila--;
                                columna++;
                                sends_pendents--;
                                reciv_pendents++;
                        }
                }
                for (dest=1; dest<num_nodes; dest++) {
                        if(reciv_pendents>0){
                                //rebem les respostes
                                reciv_pendents--;
                        }
                }
        }
        if(fila_inicial<mida_B){
                fila_inicial++;
        }else{
                columna_inicial++;
        }
        if((iteracio<mida_A)&&(iteracio<mida_B))
                mida_diagonal++;
        if((iteracio>=mida_A)&&(iteracio>=mida_B))
                mida_diagonal--;
        iteracio++;
}
```

Figura 3-17. Estructura d'organització dels nodes a MPI

Les variables **mida_A** i **mida_B** són el número de **macroblocs** en que es divideixen els fitxers A i B respectivament. **mida_diagonal** és el numero de **macroblocs** que podem executar en paral·lel a cada diagonal. **num_nodes** és el numero de nodes executats. Amb **fila** i **columna** identifiquem les coordenades del **macrobloc** a calcular. Les altres variables les usem com a comptadors necessaris per a organitzar l'execució

3.3.3 Anàlisi

A MPI tenim unes possibilitats d'escalatge més grans que a les versions de fins ara. La paral·lelització entre els nodes és total i el temps de càlcul es pot reduir tant com les dependencies ho permetin. El factor que ens limita aquí és el pas de dades necessari, que és correspon a:

- Identificador fila i columna del **macrobloc**: Del node principal als **workers**.
- Dades inicials primera fila i columna: Del node principal als **workers**.
- Dades finals última fila i columna calculades: D'un **worker** al node principal.

Per aquest motiu el que ens interessarà analitzar és les relacions de cost en temps entre l'estalvi de temps de càlcul i el temps que dediquem al pas de dades entre els nodes. D'una altre manera el que interessa es prendre mesures i fer proves per a diferents mides de **macroblocs** per triar una bona mida de tasca que doni un bon rendiment.

Una bona mida serà la que doní el millor equilibri entre:

- Ràpid aprofitament de la paral·lelització (blocs més petits).
- Millor relació comput-missatges (blocs més grans).

3.4 L'ús del paradigma MapReduce

MapReduce és una proposta per al càlcul en aplicacions que tracten grans volums de dades que proporciona moltes avantatges. Entre elles les que ens semblen més interessants per al nostre problema són la gestió, automàtica i optimitzada, del pas de dades entre els nodes, i de la execució. Aquesta està pensada expressament per a la execució de programes amb volums grans de dades en múltiples màquines.

Per a poder fer us d'aquest paradigma cal poder adaptar el problema tenint en compte com funcionen les operacions **map** i **reduce**, i com les podem usar per a fer el càlcul que a nosaltres ens interessa, de manera que després puguem recuperar el camí que ens dona l'alineament òptim. Un cop fet això hem de poder fer els càlculs respectant les dependències.

MapReduce optimitza el càlcul paral·lel repartint dades i tasques entre els diferents nodes de manera que tots els càlculs pugin efectuar-se alhora.

3.4.1 Anàlisi del problema

L'anàlisi de **MapReduce** efectuat i l'estudi dels algoritmes estudiats fa concloure que aquest paradigma no dona bones formules per a resoldre el nostre problema. El punt insalvable és el de la dependència de dades.

Ja em dit que **MapReduce** distribueix dades i tasques entre nodes per a fer el càlcul. En els algoritmes que estudiem, però, els càlculs de cada una de les diagonals depèn dels valors de l'alineament per a punts de les diagonals anteriors. Així en un primer instant només podríem calcular el valor per a l'alineament de la primera fila i columna. En següents iteracions podríem anar calculant cada diagonal amb una nova execució dels cicles **map** i **reduce** però farien falta tantes execucions com llarga sigui la més gran de les cadenes a alinear.

Vista la problemàtica, anem ara a estudiar com ho podríem fer per resoldre els alineaments òptims segons els algoritmes **S&W** i **N&W** usant el paradigma **MapReduce** i quins problemes té.

Execució per blocs:

Sembla una bona solució. Per a no tenir que fer tantes execucions del cicle **map** i **reduce** podem crear en la fase **map** parells de valors amb les dades necessàries per a poder calcular tot un bloc d'alineament tal com ho fem en les anteriors solucions. Aleshores la fase **reduce** calcularia la matriu d'alineament i crearia els fitxers de resultats.

Problemes de la execució per blocs:

Encara que ho féssim per blocs, aquests s'han d'executar en l'ordre de dependències vist, i fins a obtenir els resultats d'uns no en podem calcular uns altres. El pitjor és que per al càlcul dels blocs podem necessitar els resultats dels anteriors i estaríem obligats a llegir dels fitxers per buscar les dades necessàries.

Vistes les limitacions que ens han obligat a deixar les implementacions convencionals de **MapReduce** com a bona alternativa a **MPI**, en la recerca per millorar el rendiment per a les nostres aplicacions, ens proposem trobar altres implementacions que resolguin algunes de les limitacions. Estudiem així una versió que soluciona el problema de les dependències de dades. Ho fa permetent usar les dades obtingudes en una execució de **MapReduce** per a següents execucions, cosa que dota al paradigma de iterativitat.

3.5 MapReduce iteratiu

Twister MapReduce és la versió iterativa de **MapReduce** que hem triat per avaluar la manera d'implementar els algoritmes **N&W** i **S&W** aprofitant els avantatges d'implementar algoritmes en aquest paradigma.

Les diferencies principals respecte la implementació de **Hadoop**, i que en interessen per a aquest projecte són les que donen a aquesta versió la possibilitat de recuperar les dades després d'un cicle d'execució **map** i **reduce**. Aquestes dades les podem usar per a decidir si hem acabat un càlcul o com a entrada per a un nou cicle.

3.5.1 Anàlisi del problema

La proposta de solució segueix la idea de la execució en cicles propagant les dades dels resultats segons les dependències. A cada cicle es podrien calcular els valors d'una diagonal de dades o de blocs de dades. Els resultats, gràcies als mètodes que afegeix **twister**, poden ser recuperats i usats al següent cicle per a la següent fase de **map**. Veiem a la **[Figura 3-18]** aquesta idea.

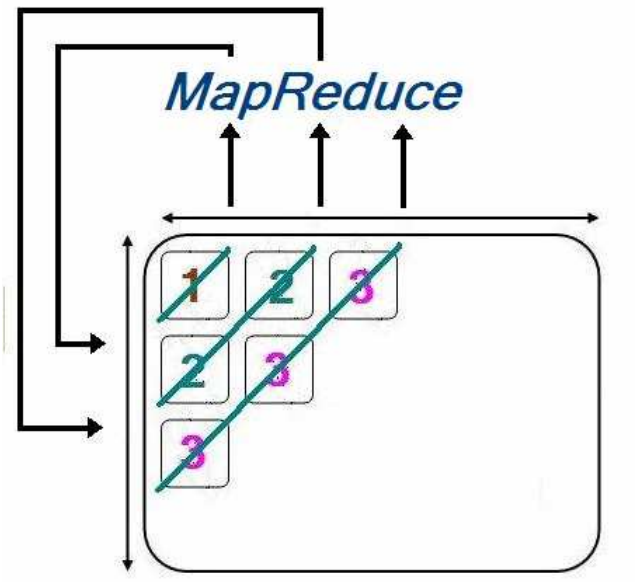

Figura 3-18. Planificació amb MapReduce Iteratiu de N&W i S&W

Problemes amb Twister MapReduce:

Encara que la idea de solució sembla factible la seva implementació és molt complexa.

En l'anàlisi que s'ha fet per a aquest projecte les principals problemàtiques que hem detectat són:

- Independència de l'ordre en les execucions de **reduce**
- Numero fixa de **maps** i **reduce** entre iteracions a **Twister**
- Mida variable del problema entre els cicles
- Cerca de dades necessàries a cada iteració entre els resultats

Possibles solucions als problemes:

- Per resoldre la independència en l'ordre de les execucions del **reduce** cal que al **map** totes les parelles **clau-valor**, que representen cada punt de la matriu de resultats, guardin les coordenades de la seva posició.
- En numero de **maps** i **reduces** fixa no té solució usant **twister**. Només podem fer que usar sempre els mateixos encara que el problema sigui de mida variable entre iteracions.
- Que la mida del problema evolucioni creixent i després decreixent és un problema a la hora de configurar els paràmetres de **twister** i obliga a organitzar les execucions pensant en el mida màxima a que arribaríem.
- La cerca de les dades entre iteracions és necessària, ja que amb els mètodes que ens permeten recuperar els resultats de la anterior iteració rebem tota la matriu, i només necessitem la última fila i columna. Cal a més que tots els valors siguin identificables amb la posició que ocupen en els resultats.

Donats els problemes identificats, ens hem vist obligats a deixar la implementació de la solució en MapReduce, que encara que possible, és molt dificultosa.

La diferència entre aquest problema i d'altres trivialment adaptables a MapReduce està en les dependencies, la necessitat de mantenir l'ordre de les dades i l'evolució de la mida entre cicles.

3.6 El monitoreig i test en totes les fases

El monitoreig **[7]** d'execucions a les diferents fases o propostes d'implementació efectuades a consistit, almenys principalment, en la inclusió de codi per a la medició de temps en temps d'execució. Aquestes medicions s'han fet tenint en compte tant criteris d'estructuració dels algoritmes com d'execució en les màquines. Així, per a cada implementació, s'han mesurat els temps que en cada test han trigat a efectuar-se un conjunt d'operacions que formen una fase.

Les fases en que dividim les tasques per a les diferents solucions proposades, poden ser:

- Inicialització de les dades
- Càlcul de l'alineament
- Guardat dels resultats
- Lectura dels fitxers d'entrada
- Planificació del càlcul

Els tests efectuats consisteixen en diferents execucions, de les que en calculem les necessitats de memòria, i per a les que mesurem els temps, tant del total del càlcul com de les fases interessants en cada cas.

S'efectuen execucions de les diferents implementacions variant els paràmetres que permetin comparar-les entre elles i valorar els guanys obtinguts en cada una.

Capitol 4 Experimentació realitzada i resultats obtinguts

Encara que hi ha certa relació entre els experiments a les diferents fases creiem interessant separar les explicacions per a cada implementació. Així per a cada una d'elles comentarem els resultats i afegirem els punts necessaris per incloure les comparacions entre elles

4.1 Experimentació i resultats a la versió Sèrie

La experimentació a aquesta versió inclou:

- Execucions per veure els temps que es triga a cada fase per a diferents mides de bloc
- Execucions per veure el que triguen les diagonals de blocs a ser calculades segons el nombre de blocs a calcular i les mides d'aquests.
- Execucions per veure el temps que triguem a calcular la matriu d'alineament entre dues seqüències segons la mida del bloc i les mides de les entrades

Dels anàlisis del temps que dedica l'execució a cada fase hem tret els resultats de la **[Figura 4-1]**. En ella veiem que el temps per inicialitzar les dades no es notable i que la proporció entre temps dedicat al càlcul respecte al temps dedicat a guardar els resultats és molt petit.

Figura 4-1. Esquema dels temps dedicats a cada fase

Els resultats de les proves per veure la influència de la mida dels blocs mostres que no hi ha cap efecte notable sobre els temps. Tampoc hi ha cap efecte si mirem la quantitat de carrega, o numero de blocs, a calcular. Això en fa pensar que amb la solució de treballar amb blocs i calculant-los un a un no carreguem la màquina i fem la mateix tasca sigui quina sigui la mida dels blocs.

Les més significatives de les dades d'aquestes proves es resumeixen en el gràfic de la **[Figura 4-2]**. Aquest representa el temps que triguem a calcular l'equivalent a un bloc de **256*256** segons la mida dels blocs utilitzats (colors blau, rosa i groc) i segons la mida de blocs per calcular (eix "**x**") en aquella diagonal. S'aprecien alguns "pics" amb els temps de càlcul per als blocs de 256 que hem comprovat no són representatius.

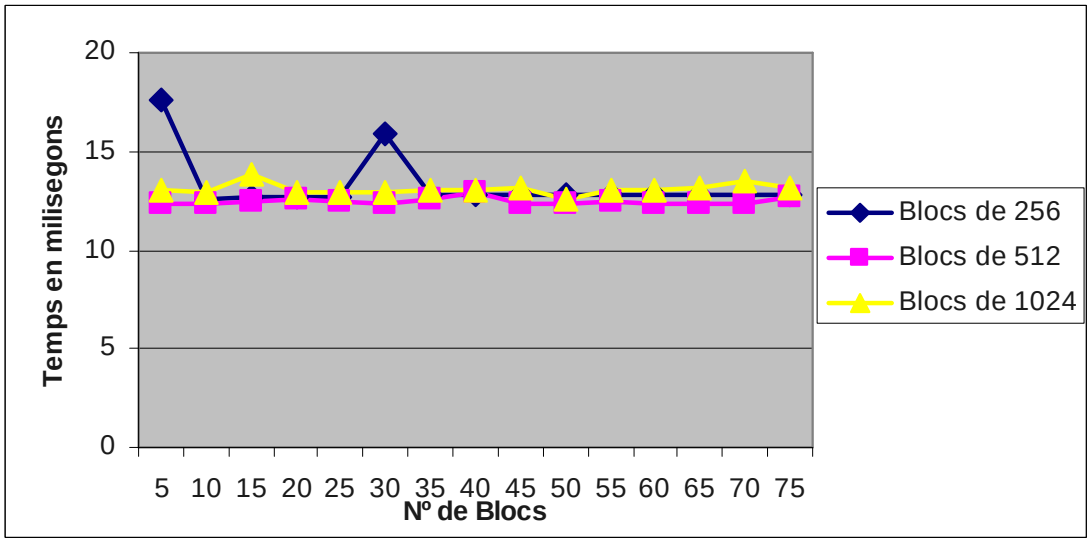

Figura 4-2. Gràfic de temps per al càlcul segons número de blocs

En la mateixa línea que l'anterior gràfic són els resultats segons la mida de les entrades. Els temps segons les entrades augmenten seguint una funció exponencial, com veiem a la **[Figura 4-3]**, i ho fa sense gaires diferencies si canviem la mida dels blocs.

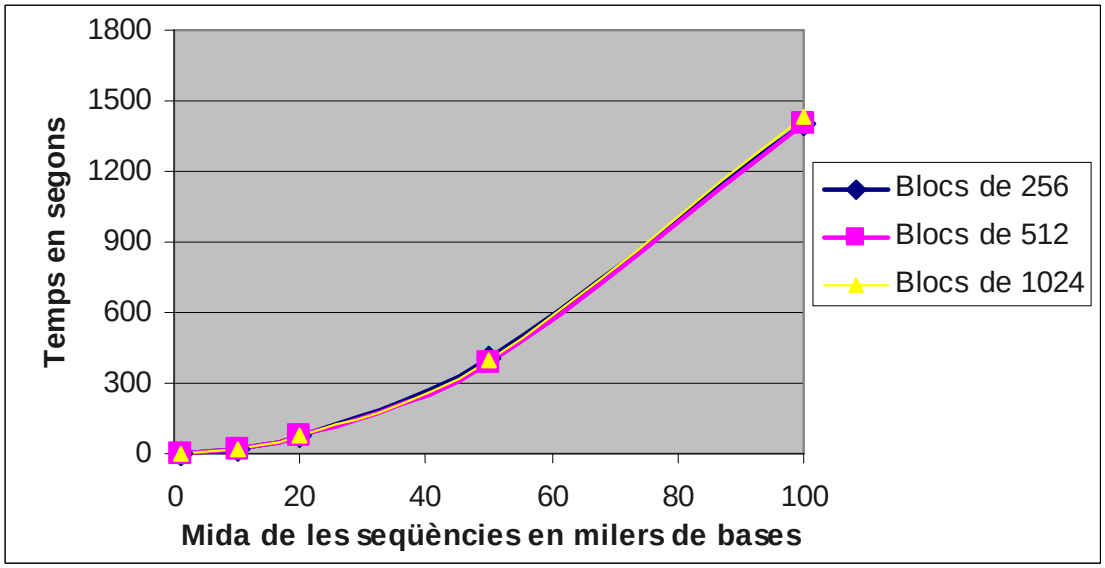

Figura 4-3. Gràfic de temps per a alineaments segons mida d'entrada

Així els resultats per a la versió sèrie mostren que podem obtenir alineaments per a seqüències gran s però que els temps per al càlcul creixen seguint una corba exponencial amb la mida de les entrades . També observem que aquests no es veuen afectats segons les mides dels blocs amb que treballem.

4.2 Experimentació i resultats a la versió usant Threads

La experimentació a aquesta versió inclou:

- Execucions per veure els temps que es triga a cada fase per a diferents mides de bloc
- Execucions per veure el que triguen les diagonals de blocs a ser calculades segons el nombre de blocs a calcular i les mides d'aquests.
- Execucions per veure el temps que triguem a calcular la matriu d'alineament entre dues seqüències segons la mida del bloc i les mides de les entrades
- Comparatives de rendiment amb la versió sèrie.

Amb els resultats de l'experimentació en aquest apartat volem trobar els paràmetres amb el millor guany en temps d'execució possible.

Les proves de rendiment amb blocs de diferents mides mostren, com s'observa a la **[Figura 4-4]**, que ara si que influencia la mida de la tasca per a cada **thread** en els temps per a realitzar els càlculs. També si observa que en créixer el nombre de blocs a operar en un cicle els temps no varien notablement.

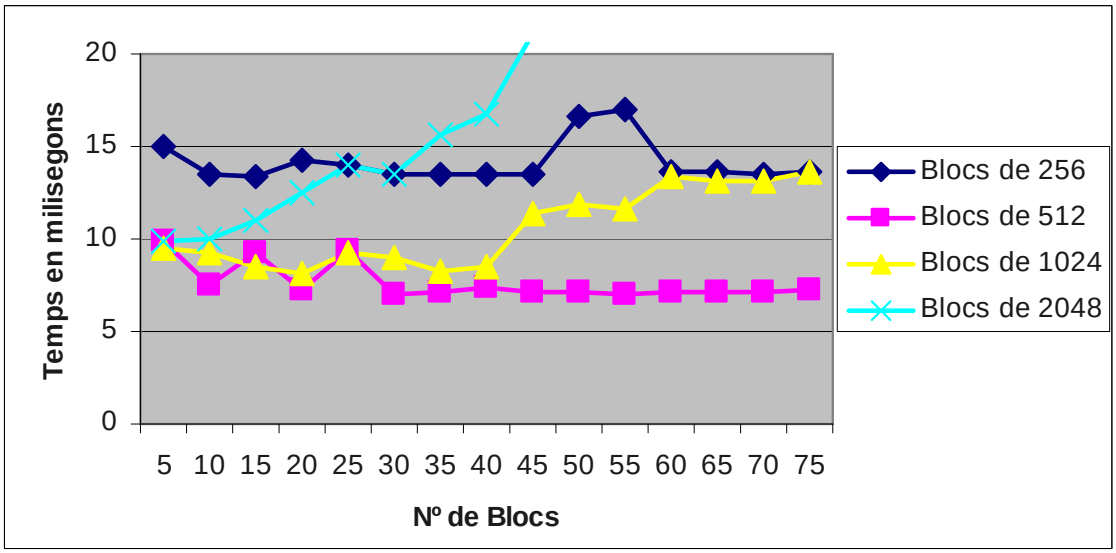

Figura 4-4. Gràfic de temps per al càlcul segons nº de blocs

Les gràfiques de la **[Figura 4-4]** representa els temps que triguen a ser calculats l'equivalent a blocs de **256*256** variant la mida real dels blocs i el nombre de blocs a calcular en el mateix cicle. Del gràfic podem concloure que de les diferents mides de bloc provades la millor és **512**.

Si mirem els temps que han trigat a ser calculats els alineaments per a seqüències de diferents mides veiem que també són menors usant blocs de **512*512**. Diferents mesures d'aquestes respostes es representen a la **[Figura 4-5]**.

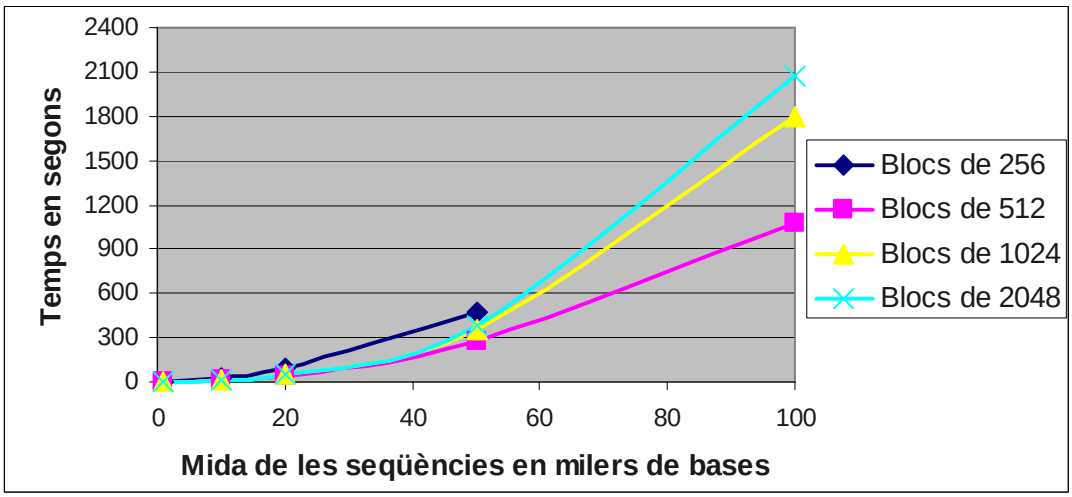

Figura 4-5. Gràfic de temps per a alineaments segons mida d'entrada

Un cop triada **512** com a la millor mida per als blocs podem comparar els resultats de les dues versions de la implementació dels algoritmes.

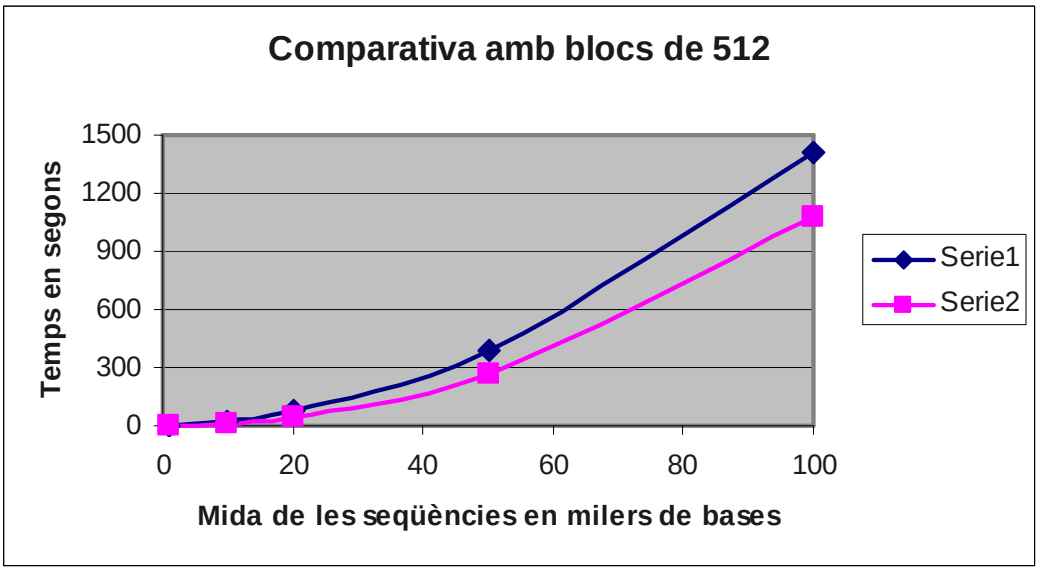

Figura 4-6. Gràfic de comparativa de temps entre les versions sèrie i threads

La **[Figura 5-6]** representa la comparació dels temps de càlcul dels alineaments, usant els diferents paradigmes.

Veiem com el rendiment utilitzant **threads** és més alt però que el guany no és tant gran com podríem esperar. El motiu és que en realitat no estem fent tots els blocs en paral·lel sinó que només solapem els càlculs per aprofitar els temps en que el processador queda lliure. Fent els càlculs hem obtingut que triga de mitjana un 28% menys i que el recurs que limita és la velocitat en les operacions d'E/S.

4.3 Experimentació i resultats a la versió usant MPI

Per a aquesta versió del codi ens interessa experimentar amb:

- Execucions per a definir una mida de **macrobloc** amb una bona relació entre paral·lelització i proporció treball i pas de missatges.
- Execucions per veure la influencia del nombre de nodes en els temps d'execució..

.

Els resultats per a la definició de la mida dels **macroblocs** mostren que no hi ha una mida bona i que aquests milloren com més grans són. Per a cada **macrobloc** hem d'enviar les dades inicials al node que l'executarà i aquest haurà d'enviar els resultats. Com menys comunicacions millors són els temps però no aconseguim treball paral·lel.

A la **[Figura 4-7]** mostrem els resultats de les mesures de temps per al càlcul de l'alineament de dues seqüències de 10.000 bases usant les tres implementacions. Les execucions usant MPI utilitzen únicament un node **master** i un node **worker** per veure com afecten als temps les comunicacions.

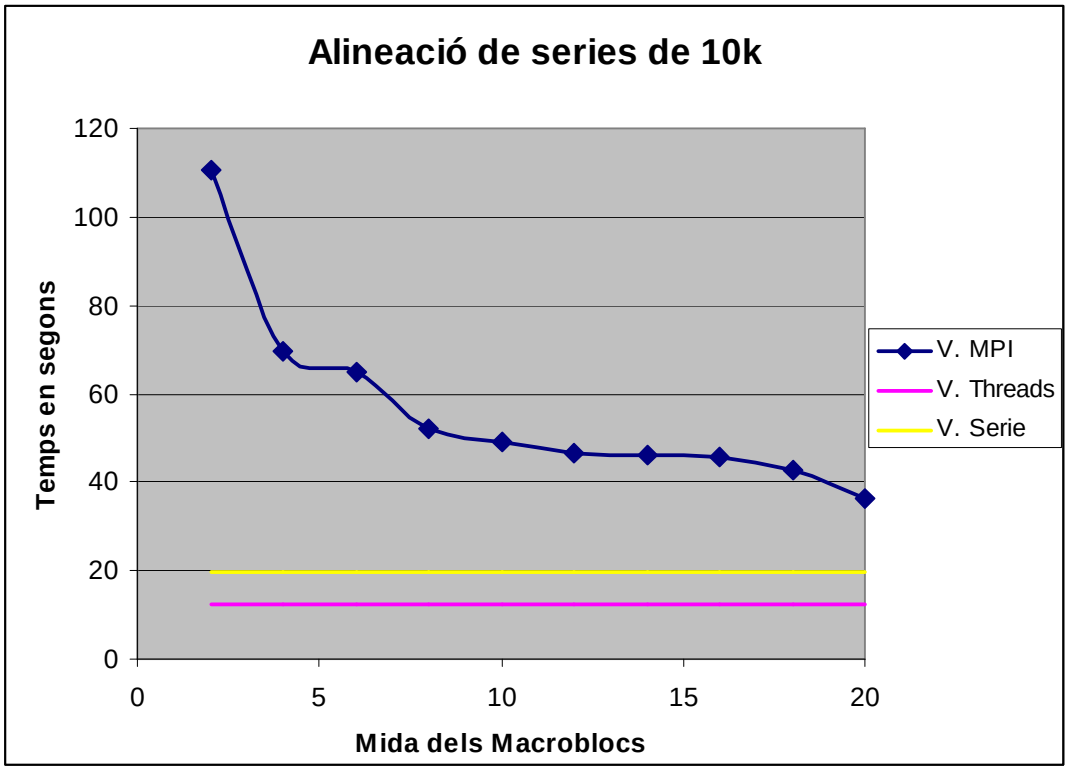

Figura 4-6. Gràfic de comparativa de temps entre les versions sèrie i threads

Veiem a la figura que de no tenir nodes disponibles la mida dels **macroblocs** influeix notablement en el temps d'execució. A mides més grans menors comunicacions i millors temps.

En segon lloc els temps varien segons el número de nodes disponibles. A més nodes més **macroblocs** podem executar en paral·lel.

Els resultats semblen contradir-se ja que mides del **macroblocs** més grans volen dir menys treball en paral·lel, per tant menys necessitat de nodes. Els resultats obtinguts no permeten arribar a conclusions clares de valors adequats que donin els millors resultats possibles en equilibri entre comunicacions i paral·lelització.

La influència del nombre de nodes és més gran com més grans són els fitxers. Els temps són menors als de les anteriors versions sempre que els temps de les comunicacions siguin menors als del càlcul de l'alineament a calcular el node.

Capitol 5 Conclusions

Dividim les conclusions en cinc apartats: aquelles referides als algoritmes, als resultats, al projecte en general, a les referides a les diferències respecte la planificació inicial i un últim per a les vies obertes on proposem nous camps d'estudi.

Conclusions sobre els anàlisis dels algoritmes

Estudiem el problema d'alinear dues seqüències, de longituds m i n, usant **S&W** i **N&W**. Si seguim l'algoritme bàsic, l'obtenció del resultat té una complexitat d'ordre **nxm** tant en espai (us de memòria) com en temps (unitats de temps dedicat). De l'anàlisi efectuat obtenim que, en el cas ideal, podem efectuar els càlculs amb complexitat **n+m**, si a cada cicle operem una diagonal sencera de les que propaguen les dependències. Es pot entendre observant la **[Figura 5-1]**, la quale mostra l'esquema de la execució en paral·lel.

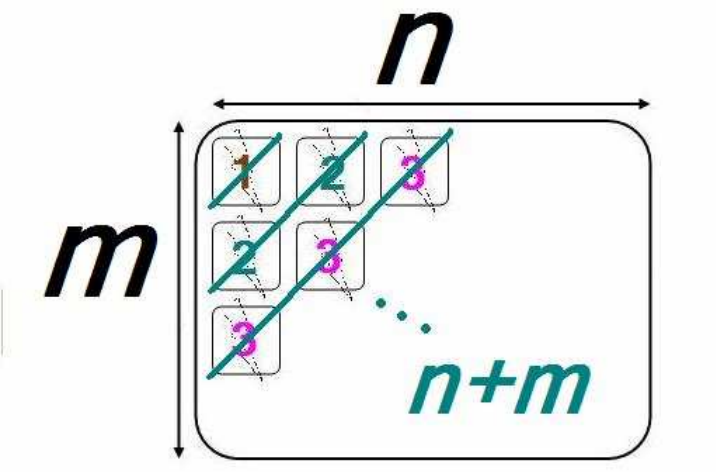

Figura 5-1. Esquema d'execució en paral·lel.

Sobre l'ordre de complexitat en espai no podem obtenir millores en el sentit de que cal guardar totes les dades calculades per a la propagació endarrere. El que si que podem millorar és la quantitat de dades que mantenim en memòria de treball, guardant a disc les que ja no necessitem.

Conclusions sobre els anàlisis dels resultats

La implementació **sèrie** permet fer els càlculs sense limitacions en la mida de les seqüències a alinear. Treballant amb blocs la màquina no ha de carregar estructures massa grans i podem operar alineaments molt grans tot i que triguem massa.

Em pogut comprovar amb els resultats que els temps milloren si fem ús del paral·lelisme, però que no aconseguim escalar tant com teòricament seria possible.

Amb l'ús de **threads** millorem els temps en aproximadament un **28%** de mitjana respecte la versió sèrie.

La solució MPI podria permetre un treball paral·lel molt bo però els temps d'enviament de dades són força grans respecte els temps que triguem en operar-los als nodes; però proporcionalment es redueixen amb la mida dels **macroblocs**.

Amb **threads** explotem la possibilitat d'executar diferents tasques dins la mateixa màquina aprofitant els temps en que s'accedeix a memòria per a fer càlculs i usa memòria compartida cosa que dona bons rendiments.

Els nodes a MPI obliguen a treballar amb memòria distribuïda de manera que s'han de passar les dades entre els nodes cosa que ens dona un factor limitant molt gran.

Analitzant els resultats obtinguts veiem que hi ha factors configurables que afecten als resultats ja que influencien directament en les quantitats de càlcul paral·lel, de dades que mantenim en memòria i de dades que s'envien als nodes a **MPI**. Aquests factors són:

- Mides de bloc: Com més petites més ràpid treballem en paral·lel però més processos s'han de crear o menys càlcul fa cada procés.
- Mides de **macrobloc**: Mides més grans donen millor relació de còmput respecte temps de pas de missatges però mantenen més temps els altres nodes inactius.
- Mides dels fitxers: Afecten només relativament. Segons la mida dels fitxers pot ser més beneficiós tenir blocs més grans o més petits. Blocs molt grans amb fitxers petits aprofiten poc la capacitat de càlcul paral·lel.

Conclusions sobre el PFC

Hem obtingut bons resultats dels anàlisis dels algoritmes. Aquests ens han permès passar d'un problema de complexitat temporal de n*m a un de complexitat n+m.

Amb les implementacions obtingudes, de tenir il·limitats nodes i comunicacions instantànies, podríem obtenir bones aproximacions a la complexitat temporal teòrica. A la pràctica, el factor limitant és que la diferència entre el temps de càlcul i d'accés a memòria és molt diferent (a raó de **1** de càlcul per **7** d'accés a memòria aproximadament).

Respecte a la planificació inicial

Hem complert en general amb la planificació proposada al inici. No s'ha complert en el punt referit a la implementació usant **MapReduce** dels algoritmes. En lloc d'això hem realitzat un anàlisi dels motius que ens han obligat a deixar-ho i de possibles alternatives.

Vies obertes i línees a seguir i millores a tenir en compte

La primera de les vies obertes que deixem és la possibilitat de treballar en la implementació en **Twister MapReduce** dels algoritmes en que hem treballat. Durant el treball hem parlat dels punts que dificulten aquesta tasca però creiem que podria ser viable, encara que costós.

També deixem per fer proves sobre les implementacions usant **Threads** i **MPI** que són interessants. Aquestes consisteixen en donar mides variables de bloc i **macrobloc** segons les dades calculables en paral·lel. Aquestes podrien ser més petites per a tasques petites.

El control de mida de les tasques es podria fer segons una sèrie de paràmetres. Podem establir mida mínima i màxima de tasca.

Els objectius d'aquestes mesures són millorar la velocitat en que aprofitem el paral·lelisme i no carregar massa la memòria de treball de les màquines. Volem algoritmes més complexes que controlin de forma dinàmica aquells paràmetres que influencien en els temps d'execució.

Una tercera via per millorar els resultats consistiria en aprofitar que el càlcul de cada punt de la matriu és molt senzill per fer-lo usant hardware i deixar pel software la planificació. Això sí, aquesta via no millora el més gran dels problemes, que és el temps dels accessos a memòria.

Capitol 6 Bibliografia

[1] “National Center for Biotechnology Information, U.S. National Library of Medicine”
http://www.ncbi.nlm.nih.gov/

[2] Web oficial. “Apache Hadoop Map Reduce”
http://hadoop.apache.org/mapreduce/ Última edició 18-07-2009

[3] Kadin Tseng. “MPI Tutorial”
http://www.bu.edu/tech/research/training/tutorials/mpi/ Última edició 08-07-2009

[4] Web oficial. “Open MPI: Open Source High Performance Computing”
www.open-mpi.org Última edició 04-06-2010

[5] Explicació, referències i exemples sobre MapReduce a Wikipedia.
(http://es.wikipedia.org/wiki/MapReduce) Última edició 15-03-2010

[6] “Bloc del farmaceuticocatracho” Última edició 22-01-2009
http://farmaceuticocatracho.wordpress.com/2009/01/22/el-genoma-humano/

[7] Keith Winston. “Midiendo el rendimiento del sistema con SAR” Última edició 03-06-2006
http://mundogeek.net/traducciones/midiendo-el-rendimiento-del-sistema-con-SAR.htm

[8] Microsoft Connect. “Dryad and DryadLINQ”
http://connect.microsoft.com/dryad

[9] Microsoft Corporation. “Dryad and DryadLINQ for Data Intensive Research”
http://research.microsoft.com/en-us/collaboration/tools/dryad.aspx

[10] Tahir Naveed1, Imitaz Saeed Siddiqui2, Shaftab Ahmed3. “Parallel Needleman-Wunsch Algorithm for Grid”. Bahria University Islamabad
http://bci.edu.pk/cse.aspx Última edició 08-06-2010

[11] Blaise Barney, Lawrence Livermore National Laboratory. “POSIX Threads Programming”
https://computing.llnl.gov/tutorials/pthreads/ Última edició 13-01-2010

[12] Indiana University. “Twister: Iterative MapReduce”
http://www.iterativemapreduce.org/ Última edició 16-03-2010

[13] UC Santa Cruz, “Genome Bioinformatics”
http://genome.ucsc.edu/

[14] Solange Galá, Analista de Sistemas, C. Del U. Entre Ríos, Argentina “Engenieria de software”
http://www.monografias.com/trabajos5/inso/inso.shtml

- Papers:

[**P1**] Jaliya Ekanayake, Xiaohong Qiu, Thilina Gunarathne, Scott Beason i Geoffrey Fox.
Pervasive Technology Institute, School of Informatics and Computing, Indiana University
"High Performance Parallel Computing with Cloud and Cloud Technologies"
http://jaliyacgl.blogspot.com/2009/06/high-performance-parallel-computing.html

[**P2**] Jaliya Ekanayake, Thilina Gunarathne i Judy Qiu.
"Cloud Technologies for Bioinformatics Applications"
ACM New York, NY, USA 2009 ISBN:978-1-60558-714-1

[**P3**] Reynaldo Gil-García y José Manuel Badía-Contelles.
"Biblioteca para el manejo automático de los buffers de comunicación en MPI"
XV Jornadas de Paralelismo, Almería 2004 ISBN 84-8240-714-7 , pags. 60-65

[**P4**] Fa ZHANG, Xiang-Zhen QIAO i Zhi-Yong LIU.
Institute of Computing Technology, Chinese Academy of Sciences, Beijing
"A Parallel Smith-Waterman Algorithm Based on Divide and Conquer"
EEE Computer Society Washington, DC, USA 2002 ISBN:0-7695-1512-6

[**P5**] Tahir Naveed, Imtiaz Saeed Siddiqui, Shaftab Ahmad.
"***Parallel Needlman Wunsch Algorithm for Grid***"
Pak-US Symposium on High Optical Networks and Enabling technologies 2005 Islamabad.

[**P6**] J. Dean and S. Ghemawat.
"Mapreduce: Simplified data processing on large clusters,"
ACM Commun., vol. 51, Jan. 2008, pp. 107-113.

ANNEXOS

Annex 1 Instal·lació i utilització dels entorns

Instal·lació de Open MPI

1. Cal tenir instal·lat C i C++

2. Descarregar la última versió de Open MPI de

http://www.open-mpi.org/software/ompi/v1.3/downloads/openmpi-1.3.3.tar.bz2

3. Extreure'n el contingut, per exemple a openmpi-1.3.3.

4. Executar:

```
$ cd openmpi-1.3.3
$ ./configure –prefix=/usr/local
```

(s'instal·larà a /usr/local)

```
# sudo make all install
```

Compilació amb Open MPI

En lloc de:

```
$ g++ my_mpi_application.cpp -o my_mpi_application
```

posarem:

```
$ mpiCC  my_mpi_application.cpp -o my_mpi_application
```

Execució amb Open MPI

Usarem els comandaments mpirun o mpiexec;

```
$ mpirun -np 4
my_parallel_application
```

(4 processos executaran 4 copies del executable "my_parallel_application)

Podem indicar els nodes disponibles fent:

```
mpirun --host node0,node1,node2,node3 ...
```

Opcions d'execució amb Open MPI

--hostfile (fitxer): opció per proporcionar un fitxer amb informació sobre els nodes on executar

--host (llista de host separats per comes): opció per especificar els hosts en els que executarem l'aplicació.

--np (o -np): Opció per indicar el número de processos a iniciar.

--wdir <directori>: Opció per especificar el directori de treball de les aplicacions (s'asumeix per defecte que es l'actual).

Exemple Open MPI

L'execució d'un programa típic podria ser:

$ mpiCC codi.cpp –o a.out
$ mpirun -np 4 a.out

Resultat :

Hola des de procés 2 de 4
Hola des de procés 1 de 4
Hola des de procés 0 de 4
Hola des de procés 3 de 4

Instal·lació i ús de Hadoop MapReduce

Podem trobar una guia a: http://hadoop.apache.org/mapreduce/

Instalació i ús de Twister MapReduce

Podem troBar una guia a: http://www.iterativemapreduce.org/

Annex 2 Codis bàsics d'exemple

Els dos programes poden ser compilats amb g++.

1. NeedlemanWunsch.cpp

```
#include <iostream>

/*
// Algoritme de programació dinamica per trovar
// alineaments globals optims
// temps i espai d'ordre n*m
// Compara matrius A i B de longituds "m" i "n" amb penalització gap per forat.
*/

#define gap -1
#define match 2
#define mis -2
#define A "ACACACTAACACACTA"; // serà la fila
#define B "AGCACACA"; // serà la columna

using namespace std;

int main (int argc, char *argv[]){
   char ga = '\0';

   //inicialitzem les dades
   int i,j;
   int d=gap;
   char vA[]= A;
   char vB[]= B;
   int m,n;
   m=sizeof(vA);
   n=sizeof(vB);

   //creem la matriu inicialitzada a zeros
   int F[n][m];

   //A partir d'aqui es pot fer treball en paral·lel

   //inicialitzem files i columnes
   for(i=0;i<m;i++)
      F[0][i]= d*i;
   for(j=1;j<n;j++)
      F[j][0]= d*j;

   //omplim matriu
   int cas1,cas2,cas3;
   for(i=1;i<n;i++){
      for(j=1;j<m;j++){
         // posiblitat 1 gap vertical
            cas1=F[i-1][j]+gap;
         // posiblitat 2 gap oritzontal
            cas2=F[i][j-1]+gap;
         // posiblitat 3 alineem
            if(vA[j-1]==vB[i-1]){
               cas3=F[i-1][j-1]+match;
```

```
            }else{
                cas3=F[i-1][j-1]+mis;
            }
          // triem el cas mes favorable
          if((cas1>cas2)&&(cas1>cas3)){
             F[i][j]=cas1;
             }else{
                if((cas2>cas1)&&(cas2>cas3)){
                F[i][j]=cas2;
             }else F[i][j]=cas3;
          }
       }
    }

    //imprimeixo matriu
    cout<<endl<<"Alineament amb valors:"<<endl;
    cout<<"   -gap: "<<gap<<endl;
    cout<<"   -match: "<<match<<endl;
    cout<<"   -mis: "<<mis<<endl<<endl<<"        ";


    for(i=0;i<m;i++){
       cout<<vA[i]<<"   ";
    }
    cout<<endl;
    for(i=0;i<n;i++){
       if (i>0)cout << vB[i-1]<<" ";
       else cout<<"  ";
       for(j=0;j<m;j++){
          if (F[i][j]>=10) cout<<" ";
          if ((F[i][j]<0)&&(F[i][j]>-10)) cout<<" ";
          if ((F[i][j]>=0)&&(F[i][j]<10)) cout<<"  ";
          cout << F[i][j] << " ";
       }
       cout << endl;
    }

    //sortir
    while(ga != 'q'){
    cout << endl << "prem 'q' per sortir:";
    cin >> ga;
    }
    return 0;
}
```

2. SmithWaterman.cpp

```
#include <iostream>

/*
// Algoritme de programació dinamica per trovar
// alineaments locals optims
// temps i espai d'ordre n*m
// Compara matrius A i B de longituds "m" i "n" amb penalització "gap" per forat.
*/

#define gap -1;
#define match 2;
#define mis -1;
```

```
#define A "ACACACTAACACACTA"; // serà la fila
#define B "AGCACACA"; // serà la columna

using namespace std;

int main (int argc, char *argv[]){
   char ga = '\0';

   //inicialitzem les dades
   int i,j;
   int d=gap;
   char vA[]= A;
   char vB[]= B;
   int m,n;
   m=sizeof(vA);
   n=sizeof(vB);

   //creem la matriu inicialitzada a zeros
   int F[n][m];
   for(i=0;i<n;i++){
      for(j=0;j<m;j++){
         F[i][j]=0;
      }
   }

   //omplim matriu
   int cas1,cas2,cas3;
   for(i=1;i<n;i++){
      for(j=1;j<m;j++){
         // posiblitat 1 gap vertical
            cas1=F[i-1][j]+gap;
         // posiblitat 2 gap oritzontal
            cas2=F[i][j-1]+gap;
         // posiblitat 3 alineem
            if(vA[j-1]==vB[i-1]){
               cas3=F[i-1][j-1]+match;
            }else{
               cas3=F[i-1][j-1]+mis;
            }
         // triem el cas mes favorable
         if((cas1>cas2)&&(cas1>cas3)){
            F[i][j]=cas1;
            }else{
               if((cas2>cas1)&&(cas2>cas3)){
               F[i][j]=cas2;
            }else F[i][j]=cas3;
         }
         // si és menor de zero es queda zero
         if (F[i][j]<0) F[i][j]=0;
      }
   }

   //imprimeixo matriu
   cout<<endl<<"Alineament amb valors:"<<endl<<endl<<"        ";
   //cout<<"   -gap: "<<gap<<endl;
   //cout<<"   -match: "<<match<<endl;
   //cout<<"   -mis: "<<mis<<endl<<endl<<"        ";

   for(i=0;i<m;i++){
      cout<<vA[i]<<"   ";
```

```
    }
    cout<<endl;
    for(i=0;i<n;i++){
        if (i>0)cout << vB[i-1]<<" ";
        else cout<<"  ";
        for(j=0;j<m;j++){
            if (F[i][j]>=10) cout<<" ";
            if ((F[i][j]<0)&&(F[i][j]>-10)) cout<<" ";
            if ((F[i][j]>=0)&&(F[i][j]<10)) cout<<"  ";
            cout << F[i][j] << " ";
        }
        cout << endl;
    }

    //sortir
    while(ga != 'q'){
    cout << endl << "prem 'q' per sortir:";
    cin >> ga;
    }
    return 0;
}
```

Josep Maria Bosch Muntal

Bellaterra, 15 de Juny de 2010

Avui en dia la biologia aporta grans quantitats de dades que només la informàtica pot tractar. Les aplicacions bioinformàtiques són la més important eina d'anàlisi i comparació que tenim per entendre la vida i aconseguir desxifrar aquestes dades. Aquest projecte centra el seu esforç en l'estudi de les aplicacions dedicades a l'alineament de seqüències genètiques, i més concretament a dos algoritmes, basats en programació dinàmica i òptims: el Needleman&Wunsch i el Smith&Waterman. Amb l'objectiu de millorar el rendiment d'aquests algoritmes per a alineaments de seqüències grans, proposem diferents versions d'implementació. Busquem millorar rendiments en temps i espai. Per a aconseguir millorar els resultats aprofitem el paral·lelisme. Els resultats dels anàlisis de les versions els comparem per obtenir les dades necessàries per valorar cost, guany i rendiment.

Hoy en día la biología aporta grandes cantidades de datos que solo con la informática podemos tratar. Las aplicaciones bioinformáticas son la más importante herramienta de análisis y comparación para entender la vida y lograr descifrar estos datos. Este proyecto centra su esfuerzo en el estudio de las aplicaciones dedicadas al alineamiento de secuencias genéticas, y más concretamente a dos algoritmos, basados en programación dinámica y óptimos: el Needleman&Wunsch y el Smith&Waterman. Con el objetivo de mejorar el rendimiento de estos algoritmos para alineamientos de secuencias grandes, proponemos diferentes versiones de implementación. Buscamos mejorar rendimientos temporales i espaciales. Para lograr mejorar los resultados aprovechamos el paralelismo. Los resultados de los análisis de las versiones los comparamos a fin de obtener los datos necesarios para valorar coste, ganancias i rendimiento.

Biology is nowadays able to extract great amounts of data from which we can obtain a lot of information. Bioinformatics applications are the most important analysis tool we have to decode this data and to understand life. This project puts its effort in studying applications dedicated to genetic sequence alignment. This is done by two algorithms based on dynamic programming: Needleman&Wunsch and Smith&Waterman. The goal is to improve these algorithm's performance on very long sequences of data and to propose different implementation options taking both time and space into account. Parallel computing will be the main tool we will be using and results will be compared to assess cost, gain and throughput.