DISEÑO E IMPLEMENTACION DE ALGORITMOS DE INFERENCIA MOLECULAR CON MPI EN PYTHON

JAIRO DAVID SERRANO LATORRE

UNIVERSIDAD INDUSTRIAL DE SANTANDER
FACULTAD DE INGENIERIAS FISICO MECANICAS
ESCUELA DE INGENIERÍA DE SISTEMAS E INFORMATICA
BUCARAMANGA
2007

DISEÑO E IMPLEMENTACION DE ALGORITMOS DE INFERENCIA MOLECULAR CON MPI EN PYTHON

JAIRO DAVID SERRANO LATORRE

Trabajo de Grado para Optar por el Titulo de Ingeniero de Sistemas

Director:
Raúl Isea
Ph. D. en Ciencias Químicas

Codirector:
Alfonso Mendoza Castellanos
Bsc. en Automática y Robótica

UNIVERSIDAD INDUSTRIAL DE SANTANDER
FACULTAD DE INGENIERIAS FISICO MECANICAS
ESCUELA DE INGENIERÍA DE SISTEMAS E INFORMATICA
BUCARAMANGA
2007

A Dios,

A Mi Mamá Clara Leonor,

A Mis Abuelos, Tios y primos

y al Amor De mi vida Sandra Liliana.

AGRADECIMIENTOS

El autor del proyecto quiere expresar su más sincero agradecimiento a todas las personas y entidades que colaboraron con este proyecto de grado:

Al Ph.D. Raúl Isea, Investigador Asociado Fundación IDEA (Caracas, Venezuela), por su invaluable aporte como director del proyecto y por la enorme paciencia.

Al Bsc. Alfonso Mendoza, Profesor Titular de la Escuela de Ingeniería de Sistemas e Informática UIS, por su valiosa colaboración y creencia en las nuevas tecnologías.

Al Centro Nacional de Cálculo Científico en la Universidad de Los Andes (Mérida, Venezuela) por la pasantía en Ciencias Computacionales: Computación Paralela, y el acceso al clúster Genoma.

Al profesor Juan José Mayorga, al Departamento de Deportes UIS, a los compañeros y amigos de la Selección de Rugby UIS, a los Ingenieros Erick, Carlos Barrios, Luis Guillermo, Juan Carlos Escobar y a todos aquellos que fueron compañeros de estudio.

RESUMEN

Titulo: Diseño e Implementación de Algoritmos de Inferencia Molecular con MPI en Python. *

Autor: Jairo David Serrano Latorre **

Palabras Clave: Bioinformática, Procesamiento Paralelo, MPI, Python, Filogenia, UPGMA, Newick, Clúster, Fasta.

Descripción:

pyUPGMA es una aplicación BioInformática de Alto Rendimiento y es una muestra del trabajo interdisciplinario entre biología e informática. pyUPGMA, reconstruye computacionalmente la filogenia de un grupo de secuencias a partir del método de Agrupamiento No Ponderado con Media Aritmética, UPGMA. La filogenia es computacionalmente demandante porque el número de posibles soluciones se incrementa rápidamente a medida que aumenta el número de taxas. UPGMA es el método más simple para reconstrucción filogenética.

pyUPGMA está desarrollado 100% en el lenguaje de programación Python con la librería de paso de mensajes pyMPI, la cual nos permitió paralelizar el método UPGMA, con lo cual se logra hacer uso del poder de cómputo de los Clúster. pyUPGMA recibe como datos de entrada un archivo con secuencias alineadas en formato FASTA, luego distribuye el trabajo en cada uno de los procesadores que conforman el Clúster “Genoma” ubicado en el Centro de Cálculo Científico de la ULA (Mérida, Venezuela), finalmente obtenemos un archivo de datos en formato Newick con sus distancias evolutivas, el cual nos permitirá reconstruir el árbol filogenético en terceras aplicaciones.

---

*Trabajo de Grado

**Facultad de Ingenierías Físico Mecánicas
Escuela de Ingeniería de Sistemas e Informática

ABSTRACT

Title: Design and Implementation of Molecular Inference Algorithms with MPI in Python. *

Author: Jairo David Serrano Latorre **

Keywords: Bioinformatic, Parallel Processing, MPI, Python, Phylogeny, UPGMA, Newick, Clusters, Fasta.

Descripción:

pyUPGMA is a High Performance Computing Bioinformatic application and a simple of interdisciplinary work between Biology and Informatic. pyUPGMA reconstruct computationally phylogeny from a group of sequences through Unweighted Pair Group Method with Mean Arithmetic. Phylogeny is demanding computationally because the number of possible solutions increases quickly as taxa number also increases.

pyUPGMA is written 100% in Python Programming Language with the library pyMPI for message passing , it allow us parallelize UPGMA, it does make that we can use the power of clusters. pyUPGMA gets data in from a file with sequences in FASTA format, then it distributes work to processors around “Genoma” cluster locate at Centro de Calculo Cientifico de la ULA (Merida, Venezuela), finally, we got a file with Newick format with evolutionary distances, which will allow reconstruct the phylogenetic tree in third party applications.

---

*Trabajo de Grado

**Facultad de Ingenierías Físico Mecánicas
Escuela de Ingeniería de Sistemas e Informática

TABLA DE CONTENIDO

LISTA DE FIGURAS

LISTA DE TABLAS

LISTA DE ANEXOS

INTRODUCCION

La bioinformática es el campo de la ciencia en el cual la biología, las ciencias de la computación, y las tecnologías de la información migran para formar una simple disciplina. El objetivo final de este campo es permitir el descubrimiento de nuevos avances biológicos así como crear una perspectiva global de los principios unificados en las ciencias Biológicas.

La preocupación del Bioinformático al principio de la revolución genómica ha sido la creación y el mantenimiento de una base de datos, para almacenar la información biológica, tal como secuencias de nucleótidos y de aminoácidos. El desarrollo de este tipo de base de datos implicó no sólo problemas de diseño sino además el desarrollo de interfaces complejas por el que los investigadores pudieran acceder a dichos datos existentes. En última instancia, toda esta información se debe combinar para formar un cuadro comprensivo de actividades celulares normales de modo que los investigadores puedan estudiar por ejemplo estados de la enfermedad. Por lo tanto, el campo de la bioinformática ha evolucionado tanto que ahora las tareas envuelven el análisis y la interpretación de varios tipos de datos, incluyendo secuencias de nucleótidos y de aminoácidos, dominios de la proteína, y estructuras de la proteína.

La Bioinformática incluye:

- El desarrollo y la implementación de herramientas que permitan el acceso eficiente, para administrar y usar diferentes tipos de información.

- El desarrollo de nuevos algoritmos y estadísticas con las cuales determinar relaciones entre los miembros de un conjunto grande de datos, tales como localizar un gen dentro de una secuencia, predecir la estructura y la función de la proteína, asi como identificar familias de secuencias relacionadas entre si.

¿Por qué la bioinformática es tan importante?

Integración es la palabra clave para entender la importancia de la bioinformática, ya que a través de herramientas y utilizando la información ya depositada en bases de datos alrededor del mundo se está comenzando a descubrir relaciones no triviales escondidas en el código de la vida.

El fundamento lógico para usar avances computacionales para facilitar la comprensión de varios procesos biológicos incluye:

- Una perspectiva más global en el diseño experimental.

- La habilidad para capitalizar en la tecnología emergente de la minería de bases de datos, los procesos por el cual las hipótesis probables son generadas con respecto

de la función o estructura de un gen o proteína de interés para identificar secuencias similares en organismos mejor caracterizados. [1]

¿Qué es la filogenia?

Charles Darwin publicó el Origen de las Especies en 1859, donde proponía un mecanismo para explicar cómo se ha originado toda la diversidad de la vida. A partir de esta obra de Darwin arranca el núcleo de la teoría evolutiva actual. De manera general se puede decir:

- Los seres vivos se reproducen, pero no sobrevivirán todos sus descendientes.
- Cuando hay reproducción sexual, los hijos resultan diferentes a sus padres.
- Los seres vivos con mejor adaptación a su medio ambiente tendrán posibilidades más altas de sobrevivir.
- Generación tras generación, la naturaleza selecciona los mejor adaptados.

Como conclusión toda especie viva se ha originado a partir de otra anterior. Dado que todos los seres vivos compartimos el mismo código genético, se puede deducir que todos los habitantes del planeta desde la bacteria más simple hasta la más grande ballena azul, descendemos de un antepasado común, que habitó la tierra hace unos 4000 millones de años. Luego la filogenia estudia las relaciones evolutivas entre las distintas especies, reconstruyendo la historia de su diversificación desde el origen de la vida en la Tierra hasta la actualidad. La filogenia proporciona el fundamento para la clasificación de los organismos.

En este libro se describe el proceso para el diseño y la implementación del algoritmo para cálculos de filogenia el cual se basa en el método llamado UPGMA que son las siglas de Unweighted Pair Group Method with Arithmetic Mean que se traduce como Método de Agrupamiento No Ponderado con Media Aritmética. Este tipo de calculo puede llegar a ser muy largo por lo que se implementara una versión en paralelo del mismo. Para lograr implementar el mismo, los datos iniciales del programa busca la información necesaria provenientes de Bases de Datos Biológicas y asi realizar la Reconstrucción Filogenética por el método UPGMA (algoritmo basado en una matriz de distancias para la reconstrucción de arboles filogenéticos) empleando el lenguaje de programación Python y librerías para el cálculo en paralelo llamada pyMPI.

En el capítulo 2 y 3 corresponden al marco teórico, la metodología y desarrollo del programa, en este último se explica cómo se llevo a cabo el desarrollo secuencial del algoritmo UPGMA.

El capitulo 4 y 5 corresponden al desarrollo de pyUPGMA (parallel python UPGMA) y las pruebas realizadas para medir el rendimiento respecto de la versión secuencial, a través de Python Profiler.

En el capítulo 6 se expone el origen del Virus de Inmunodeficiencia Humana Tipo I (HIV-1) y del Virus de Papiloma Humano, además se compara el árbol filogenético obtenido por pyUPGMA con el obtenido por la herramienta Jalview.

[1] http://www.ncbi.nlm.nih.gov/About/primer/bioinformatics.html

En el capítulo 7 se exponen las conclusiones que se obtuvieron a través del desarrollo de este proyecto de investigación.

En el capítulo 8 se exponen algunas recomendaciones para mantener la investigación interdisciplinaria y recomendaciones para diseñar mejores aplicaciones Bioinformáticas de alto rendimiento.

Finalmente, los resultados presentados en esta tesis fueron presentados en el "International Workshop on Collaborative Bioinformatics RIBIO-EMBNET 2007", en Torremolinos, España del 11 al 14 de Junio. http://www.embnet.org/node/71.

1. PRESENTACION DEL PROYECTO

1.1. PLANTEAMIENTO DEL PROBLEMA

En las últimas décadas se ha incrementado el conocimiento en áreas como la Genética y la Biología gracias a avances tecnológicos logrados mediante una investigación interdisciplinaria que ha permitido establecer vínculos entre la medicina, la biología, la química y las ingenierías (especialmente las ingenierías de sistemas y electrónica). Este vertiginoso crecimiento se debe en parte al impresionante avance de las ciencias de la computación y a su aplicación directa al estudio de la genética alcanzando así logros representativos como la primera secuenciación del Genoma Humano [Myers y col; 2001].

Estos significativos progresos han generado una enorme cantidad de datos (principalmente estructuras primarias de multitud de genes de todo tipo de organismos hasta llegar a tener más de 100 genomas de organismos celulares completamente secuenciados) cuyo análisis demanda del uso de herramientas computacionales altamente especializadas. El desarrollo de estas herramientas tiene por nombre "Bioinformática" y es considerada hoy por hoy una de las grandes revoluciones en las ciencias biológicas y computacionales. Con el uso de la bioinformática ha sido posible empezar la anotación de los genomas y esta ha resultado en la elaboración de mapas genéticos y físicos cada vez más complejos, por ejemplo, en el año de 1995 se tenían únicamente 10 genomas completamente secuenciados, para 1997 se tenían 30 genomas celulares secuenciados y para el 2001 se tenían más de 800 genomas completamente secuenciados, incluyendo genomas de organismos no celulares como virus y plásmidos, a la fecha el numero de genomas secuenciados ha aumentado significativamente.

Pero el problema no termina con la primera secuenciación, por el contrario, es aquí donde se hace aún más necesario el uso de la bioinformática para que el investigador logre transformar esos datos en información útil, que permita el desarrollo de nuevas tecnologías para la elaboración de medicamentos, alimentos, pruebas de paternidad y nuevos avances en la curación de enfermedades y, en definitiva, logre una mayor comprensión de cómo funcionan los seres vivos. Con la última secuenciación del genoma humano se tendría de manera inmediata el desarrollo de diagnósticos para la detección precoz de muchas enfermedades genéticas y se conocería la base molecular de muchas de estas enfermedades. Entre las funciones que tiene que desarrollar el investigador al estudiar un genoma están la identificación de todos los genes, la determinación de su función biológica y su influencia en las diversas patologías y secuencias metabólicas que pueden afectar a un organismo. [2]

Teniendo en cuenta que la Filogenia es computacionalmente demandante: en su búsqueda por reconstruir una relación evolutiva entre tres o varios genes u organismos, y

[2] http://www.virtual.unal.edu.co/cursos/ingenieria/2001832/index.html

que además los biólogos estiman hoy en día que en la tierra viven entre 5 y 100 millones de especies de organismos, podemos representar gráficamente esta relación evolutiva en los árboles filogenéticos, los cuales pueden alcanzar millones de posibles combinaciones, (por ejemplo, un conjunto de datos de 10 genes podría obtener hasta 2'027.025 arboles posibles). Es por eso que se planteó en este proyecto el diseño y la implementación del algoritmo UPGMA Paralelo, para minimizar el tiempo de cómputo para la reconstrucción filogenética a través de los procesadores que conforman la arquitectura de computadoras en paralelo. En este sentido se emplearon el clúster "Genoma" propiedad del Centro de Cálculo Científico de la Universidad de Los Andes (Mérida, Venezuela), el clúster "SIECAR" propiedad de la Universidad Pontificia Bolivariana de Bucaramanga.

1.2. OBJETIVOS

1.2.1. Objetivo General

Diseñar e implementar, un algoritmo de procesamiento en paralelo, con base al algoritmo UPGMA (Unweighted Pair Group Method with Arithmetic Mean) usando la librería de paso de mensajes MPI (Message Passing Interface) para el lenguaje de programación Python.

1.2.2. Objetivos Específicos

- Reconocer el estado del arte y el desarrollo actual de nuevos lenguajes de programación de alto nivel implementados por las comunidades científicas y educativas por la facilidad de su uso.

- Reconocer las ventajas y desventajas del algoritmo UPGMA para cálculos de inferencia molecular en la determinación de la filogenia molecular.

- Desarrollar e Implementar el algoritmo computacional UPGMA secuencial en el lenguaje de programación Python, independientemente de la plataforma computacional para su ejecución.

- Desarrollar e Implementar el algoritmo computacional UPGMA paralelo en el lenguaje de programación Python, independientemente de la plataforma computacional para su ejecución.

- Determinar la eficiencia y el rendimiento de los algoritmos desarrollados en Python (tanto secuencial como en paralelo) con librerías especialmente diseñadas para ello como es Python Profiler (detalles en http://docs.python.org/lib/profile.html).

2. MARCO TEORICO

2.1. FILOGENIA

El alto contenido histórico de la biología moderna requiere de un lenguaje que sea capaz de expresar en forma rigurosa las relaciones genealógicas de los distintos organismos y que permita una fluida comunicación de los especialistas de las diversas ramas de esta ciencia. La reconstrucción de la historia de los organismos se llama filogenia.

La palabra filogenia está compuesta de la siguiente manera: del griego phylon=tribu, raza y -genia=relativo al origen. Según el Diccionario de la Real Academia de la Lengua Española la Filogenia es la parte de la Biología que se ocupa de las relaciones de parentesco entre los seres vivos. Literalmente se puede decir que la filogenia es el estudio de la evolución de las formas de vida, o si se prefiere una definición más elaborada, la filogenia es la disciplina que estudia las relaciones evolutivas entre las distintas especies, reconstruyendo su diversificación desde el origen de la vida en la tierra hasta nuestros días.

Fig. 2.1. Árbol de la Vida.
Fuente: http://www.tolweb.org

Las relaciones filogenéticas pueden ocurrir en diferentes niveles tales como genes, proteínas y especies. Estas relaciones son usualmente presentadas bajo la forma de arboles filogenéticos en los cuales las ramas representan los patrones de divergencia de diversos organismos o partes del organismo (ejemplo, en genes).

Un árbol filogenético es básicamente una gráfica compuesta de nodos y ramas con la cual pretendemos representar las relaciones evolutivas entre unidades taxonómicas operacionales ("OTU" por sus siglas en inglés). Los nodos representan las unidades taxonómicas, y las ramas definen las relaciones de ancestro y descendencia entre las unidades OTU. Los nodos pueden ser externos en cuyo caso representan las OTUs bajo estudio, los cuales pueden ser especies, poblaciones, genomas, individuos, o genes. O bien, los nodos pueden ser internos los cuales representan los OTUs ancestrales. Las ramas también pueden ser externas o internas. La longitud de las ramas representan normalmente el número de cambios que han ocurrido en esa rama.

Los árboles filogenéticos pueden tener raíz o no. En los árboles con raíz existe un nodo particular llamado la raíz, a partir del cual hay un camino evolutivo único que conduce a cualquier otro nodo, y representa el último ancestro común de los OTUs bajo estudio. Se dice que un árbol con raíz es un árbol polarizado, en donde conocemos la dirección de los cambios. En un árbol sin raíz solo se especifican las relaciones entre los OTUs. La mayoría de los métodos de reconstrucción filogenética generan árboles sin raíz. Para poder ponerle raíz a un árbol, normalmente es necesario utilizar un grupo externo ("outgroup", en ingles), es decir, un OTU del cual tengamos evidencia a partir de información externa tal como el registro fósil que claramente indique que dicho OTU ha ramificado antes que el resto de los OTUs bajo estudio.

El número posible de árboles con y sin raíz se calcula de acuerdo a la cantidad **n** de OTUs, de la siguiente forma[3]:

- Para calcular el número posible de árboles sin raíz:

$$N_U = (2n - 5)! / [2^{n-3} (n - 3)!]$$

- Para calcular el número posible de arboles con raíz:

$$N_R = (2n - 3)! / [2^{n-2} (n - 2)!]$$

[3] Li, Wen-Hsiung (1997) Molecular Evolution. Sinauer Associates, USA

Tabla 2.1. Número de árboles con y sin raíz para 10 OTU's.

OTU's	Árboles con raíz	Árboles sin raíz
2	1	1
3	3	1
4	15	3
5	105	15
6	954	105
7	10.395	954
8	135.135	10.395
9	2.027.025	135.135
10	34.459.425	2.027.025

De la Tabla 2.1 se puede notar que la inferencia filogenética representa un problema computacionalmente exigente a medida que se incrementa el número de OTUs y además de este gran número de árboles solo uno es el verdadero, sólo uno representa la historia evolutiva. (Además un árbol que se obtiene de un conjunto de datos mediante un método de reconstrucción de árboles es llamado un árbol inferido, este árbol puede o no diferir del árbol evolutivo real.)

Por último la inferencia filogenética se divide en dos grandes enfoques:

- La Cladística: Se puede definir como el estudio de las sendas evolutivas, es decir, la cladística intentar resolver preguntas como:
  - ¿Cuántas ramas existen entre un grupo de organismos?
  - ¿Qué ramas se conectan entre sí?
  - ¿Cuál es el patrón de bifurcación?

  Un árbol que establece las relaciones antepasado-descendientes se llama cladograma. Un cladograma se refiere a la topología de un árbol filogenético.

- La Fenética: Es el estudio de las relaciones entre un grupo de organismos con base en el grado de similitud entre ellos, la similitud puede ser molecular, por fenotipos o anatómica. Un árbol que expresa las relaciones fenéticas recibe el nombre de fenograma.

La principal diferencia entre los dos radica en que un fenograma puede servir como indicador de las relaciones cladísticas, pero no siempre un cladograma servirá de indicador de las relaciones fenéticas. Sólo en el caso en que se presente una relación lineal entre el tiempo de divergencia y el grado de divergencia genética o morfológica los dos tipos de árboles (Cladograma y Fenograma) serán iguales.

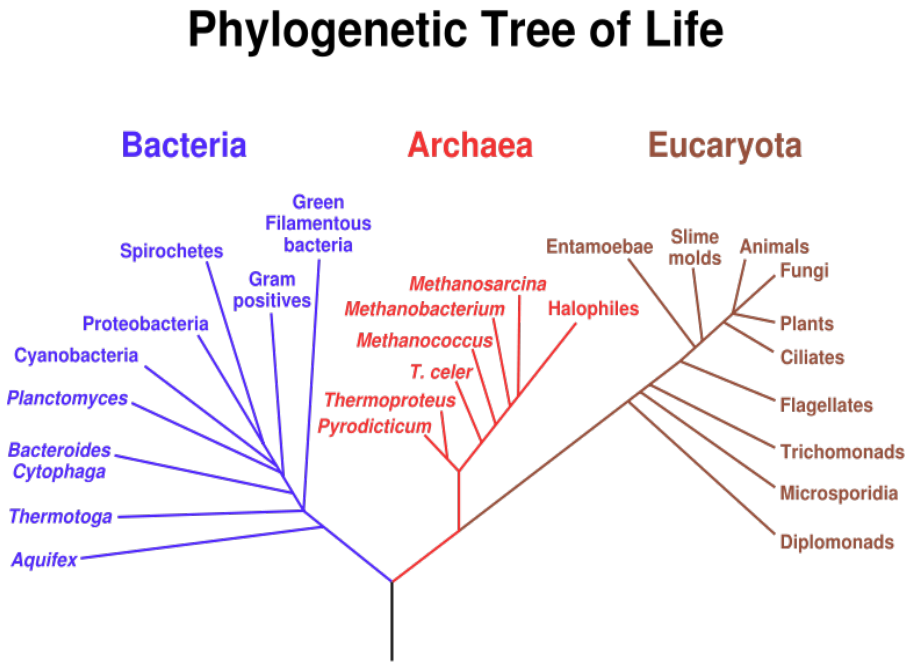


Fig. 2.2. Árbol Filogenético de la Vida.
Fuente: http://es.wikipedia.org/wiki/Árbol_filogenético

Los árboles filogenéticos se construyen tomando en cuenta la teoría de la evolución, que nos indica que todos los organismos son descendientes de un ancestro común: la protocélula. Así, todos los organismos, ya sean vivos o extintos, se encuentran emparentados en algún grado.

¿Pero por qué usar filogenia?

- Como la astronomía, la biología es una ciencia histórica.
- El conocimiento del pasado es importante para resolver muchas preguntas relacionadas a patrones y procesos biológicos.
- Es usada en diferentes áreas de la biología, desde población genética a estudios macroevolucionarios, desde epidemiología a conducta animal, desde práctica forense a conservación ecológica.
- Para predecir las características de unos organismos basadas en las características de sus similares. Entender las relaciones de los organismos con otras especies es clave para entender sus características.
- Para prevenir comparaciones impropias basadas en relaciones no existentes.

2.2. METODOS DE RECONSTRUCCION FILOGENETICA

La evolución molecular es una de las claves para comprender la información disponible en las bases de datos de biología. La filogenia molecular permite comprender la evolución de genes y genomas, así como interpretar la conservación en secuencias de ADN y proteínas. Por ello, en la actualidad constituye una herramienta fundamental en biotecnología, genómica y proteómica.

El objetivo de la filogenética molecular es convertir la información que existe entre un grupo de secuencias homólogas en un árbol evolutivo (patrón filogenético), el cual es en realidad una hipótesis de relaciones genealógicas. Es importante tener en cuenta que para un grupo dado de genes debe de existir solo una historia evolutiva real, la cual, tratamos de inferir a través de los métodos de reconstrucción filogenética, y es por eso que toda filogenia es en realidad una hipótesis evolutiva. Por ejemplo, para 9 genes habrá 135,135 árboles posibles sin raíz, y solo uno de ellos representará la historia evolutiva real, *los métodos de reconstrucción filogenética intentan encontrar ese árbol unico.*

Ya más particularmente se puede decir que los objetivos de una reconstrucción filogenética son:

a) Reconstruir correctamente las relaciones genealógicas, es decir ¿cuál es la Topología (el patrón de ramificación) correcta?
b) Estimar correctamente las longitudes de las ramas, donde la longitud de las ramas está determinado por b = rt donde (b = longitud de ramas; r = tasa de evolución; t = tiempo de divergencia)
c) Conocer la raíz del árbol (el ancestro común).

Distintos métodos de reconstrucción filogenética que existen pueden ayudar a estudiar cada uno de los objetivos anteriores, con mayor o menor precisión.

Tabla 2.2. Clasificación de los Métodos de Reconstrucción Filogenética

TIPO DE DATO			
METODO DE CONSTRUCCION		DISTANCIAS	CARACTERES
	Agrupamiento	UPGMA NEIGHBOUR JOINING	
	Criterio Optimización	MINIMA EVOLUCION	MAXIMA PARSIMONIA MAXIMA VEROSIMILITUD

La clasificación observada en la Tabla 2.2 fue planteada por Page & Holmes[4]. Según Page & Holmes los métodos se pueden clasificar de acuerdo a:

- Como utilizan los datos de las secuencias (distancias o caracteres discretos)
- Como construyen el árbol (agrupamiento o búsqueda)

Los métodos de distancias convierten las secuencias alineadas en una matriz de distancias (el número de sustituciones entre dos secuencias) y luego utilizan esa matriz para construir el árbol. Los métodos de caracteres discretos utilizan cada sitio de la secuencia (o una función de cada sitio) directamente. Los métodos de agrupamiento construyen un árbol a partir de un algoritmo determinado. Mientras que los métodos de búsqueda utilizan un criterio de optimización para decidir entre todas las posibles hipótesis (árboles evolutivos) cual es el que mejor se comporta.

Método de Mínima Evolución

Para cada árbol posible alternativo se puede estimar la longitud de cada rama a partir de las distancias pareadas estimadas (d_{ij}) entre los taxa, y luego computar la suma (S) de todos los estimados de las longitudes de las ramas. El criterio de mínima evolución es escoger el árbol con el menor valor S dado por:

$$\mathrm{S} = \sum_{i}^{T} b_i$$

UPGMA

Unweighted Pair-Group Method with Arithmetic Mean que se traduce como Método de Agrupamiento No Ponderado con Media Aritmética, es el método más sencillo de reconstrucción de árboles. Originalmente fue diseñado para la construcción de fenogramas (diagramas que reflejan las similitudes fenotípicas entre los OTU's), pero también se puede utilizar para la construcción de árboles filogenéticos sí la tasa de evolución entre los OTU's es aproximadamente constante, de tal forma que existe una relación aproximada entre la distancia evolutiva y el tiempo de divergencia. UPGMA utiliza un algoritmo de agrupamiento (clustering).

UPGMA asume:

[4] Roderic, D.M. Page & Edward Holmes (1998) Molecular Evolution A Phylogenetic Approach. Blackwell Science, United Kingdom

- Los OTU's que se están analizando evolucionaron a la misma tasa, es por esta razón que la matriz de cambios que se forme debe tener distancias ultramétricas.
- Las OTU's son mutuamente independientes, esto significa que un cambio en un determinado sitio no altera la distribución en los demás.
- Después que dos especies divergen continúan evolucionando independientemente.

¿Cómo usar UPGMA?

1) Dado cuatro OTU's A, B, C y D, se calcula el número de sustituciones entre ellas.

Tabla 2.3. Matriz de Distancias

	A	**B**	**C**
B	d_{AB}		
C	d_{AC}	d_{BC}	
D	d_{AD}	d_{BD}	d_{CD}

2) Seleccionamos la menor distancia entre los OTU's, por ejemplo se puede suponer que las OTU's que más se parecen son A y B, con estos formamos el primer grupo como se ilustra en la figura 3. Estos dos OTU's serán tratados ahora como un solo OTU compuesto.

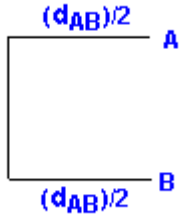


Fig. 2.3. Primera Ramificación

Modificamos la tabla 3 con los nuevos OTUs,

Tabla 2.4. Nueva Matriz de Distancias

	AB	**C**
C	$d_{(AB)C}$	
D	$d_{(AB)D}$	d_{CD}

3) Nuevamente seleccionamos el par de OTU's que presenten el menor número de cambios y los agrupamos. Se ha supuesto que estos OTU's son AB y C. ver tabla 5 y figura 4.

Tabla 2.5. Matriz de Distancias Reducida

	ABC
D	$d_{(ABC)D}$

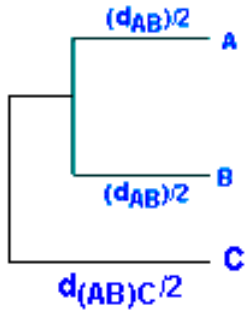


Fig. 2.4. Segunda Ramificación

4) Por último terminamos el árbol. La topología final se muestra en la figura 5.

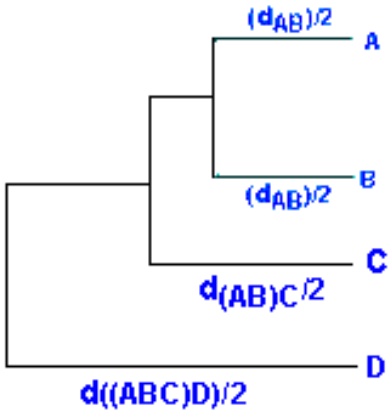


Fig. 2.5. Grupo Completo de Ramificaciones

DISTANCIAS ULTRAMETRICAS: Las distancias ultramétricas son aquellas que cumplen el criterio de los tres puntos (The Three-point condition), está condición dice:

Sean tres OTU's A, B y C se cumple la siguiente desigualdad:

$$d_{AC} <= \text{máximo } (dAB, d_{BC})$$

Por ejemplo en la figura 6 se ilustra un árbol que cumple esta condición.

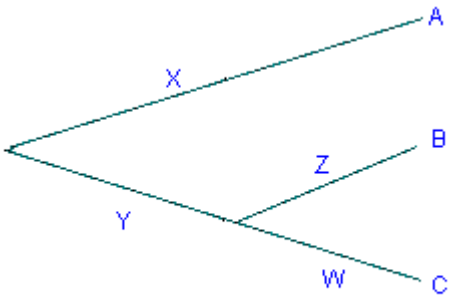


Figura 2.6. Árbol con distancias ultramétricas.

En la siguiente tabla se resumen la propiedad aditiva y ultramétrico del árbol de la figura 2.6.

Tabla 2.6. Propiedades Aditivas y Ultramétricas

Propiedades Aditivas	Propiedades Ultramétricas
$\mathbf{d_{AB} = x + y + z}$	z = w
$\mathbf{d_{AC} = x + y + w.}$	x = y + w
$\mathbf{d_{BC} = z + w.}$	x = y + z

NEIGHBOUR-JOINING

Neighbour-Joining que significa Unión de vecinos, es el método propuesto por Saitou and Masatoshi Nei en el año de 1987. Dicho metodo es la unión de los OTU's más cercanos (vecinos) tratando de minimizar la longitud total del árbol. El método inicia con un árbol en forma de estrella en el cual todos los OTU's están unidos a un nodo central. Figura 2.7. Las distancias deben cumplir la condición de los cuatro puntos conocido por las palabras en ingles: The Fourth-Point condition.

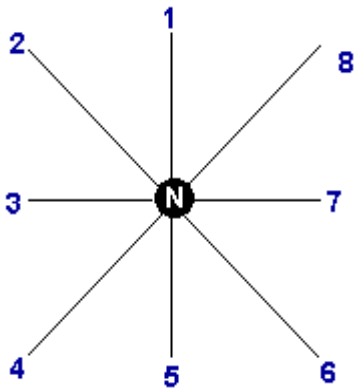


Fig. 2.7. Arbol en forma de Estrella

¿Cómo usar NEIGHBOUR-JOINING?

1) Se calcula la divergencia de la red para cada OTU, denotada con la letra r.

   $r(i) = d_{i1} + d_{i2} + d_{i3} + ... + d_{ij}$. i es cualquier OTU y j es el número total de OTU's.

2) Se calcula la nueva matriz de distancias con la siguiente formula,

$$Mij = d_{ij} - \frac{[\, r(i) + r(j) \,]}{N - 2}$$

   M_{ij} = Nueva distancia entre los OTU's i y j.
   d_{ij} = Distancia actual entre los OTU's i y j.
   r(i) = Divergencia del OTU i.
   r(j) = Divergencia del OTU j.
   N = Número de OTU's.

3) Se escogen como vecinos el par de OTU's que tengan el menor valor de M_{ij} (los más negativos)y se forma un nuevo nodo k y se procede a estimar la longitud de las ramas que unen el nodo interno K y los OTU's i y j. Estas distancias se calculan con la siguiente formula,

$$S(iu) = \frac{d_{ij}}{2} + \frac{[\, r(i) - r(j) \,]}{2\,(N - 2)}$$

$$S(ju) = d(ij) - S(iu)$$

4) Se estiman las distancias del resto de OTU's al nodo interno k. Estas distancias se calculan así,

   Sean vecinos los OTU's i y j en el nodo k y sea n un OTU, entonces la distancia de nodo interno k al OTU es:

$$d(nk) = \frac{d(in) + d(jn) - dij}{2}$$

5) Se reduce en 1 el número de OTU's N y se inicia de nuevo el proceso, mientras que N sea mayor que 2.

DISTANCIAS ADITIVAS: La distancia evolutiva entre cada par de OTU's deberá ser igual a la suma de las longitudes de las ramas que las unen, en la figura 2.8 se puede ver un

ejemplo de un árbol con distancias aditivas. Es de notar que las longitudes de las ramas también representan las distancias evolutivas entre un par de OTU's.

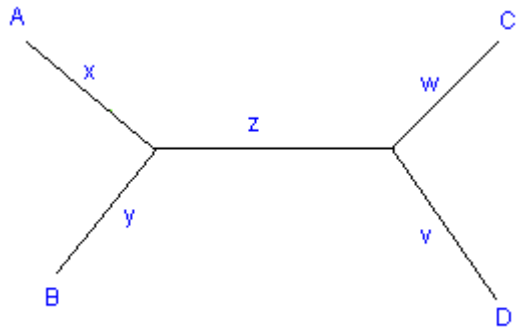


Fig. 2.8. Árbol Aditivo

Entonces las propiedades aditivas del árbol son:

$d_{AB} = x + y$
$d_{AC} = x + z + w$
$d_{AD} = x + z + v$
$d_{BC} = y + z + w$
$d_{BD} = y + z + v$
$d_{CD} = w + v$

Un árbol con distancias aditivas cumple **la condición de los cuatro puntos** ("The four-point condition", Buneman, 1971), este criterio dice que si se tienen cuatro (4) OTU's A, B, C y D se cumple:

$$d_{AB} + d_{CD} <= \text{máximo} (d_{AC} + d_{BD}, d_{AD} + d_{BC})$$

Aplicando este criterio a la figura 2.8 obtenemos:

$d_{AB + dCD} = (x + y) + (w + v) = x + y + z + w$.
$d_{AC} + d_{BD} = (x + z + w) + (y + z + w) = x + y + 2z + 2w$.
$d_{AD} + d_{BC} = (x + z + v) + (y + z + w) = x + 2z + v + w$.

entonces:
$d_{AB} + d_{CD} <= \text{máximo} (d_{AC} + d_{BD}, d_{AD} + d_{BC})$
$x + y + z + w <= \text{máximo} (x + y + 2z + 2w, x + 2z + v + w)$

MAXIMUM PARSIMONY

Maximum Parsimony - MP (Máxima Parsimonia), referido a menudo simplemente como Parsimonia es un método estadístico no paramétrico. Bajo el criterio de máxima parsimonia el árbol filogenético inferido es el árbol que requiera el menor número de cambios evolutivos. El criterio de optimización son las restricciones y permisos otorgados a los caracteres para que sus estados puedan cambiar.

En general, los métodos de parsimonia (Wagner, Fitch, Dollo) operan seleccionando árboles que minimizan la longitud total del árbol. En términos matemáticos: del conjunto de árboles posibles, encuentre todos los arboles τ para el cual $L_{(\tau)}$ es mínima.

$$L_{(\tau)} = \sum_{k=1}^{B} \sum_{j=1}^{N} w_j \cdot diff(x_{k'j}, x_{k''j})$$

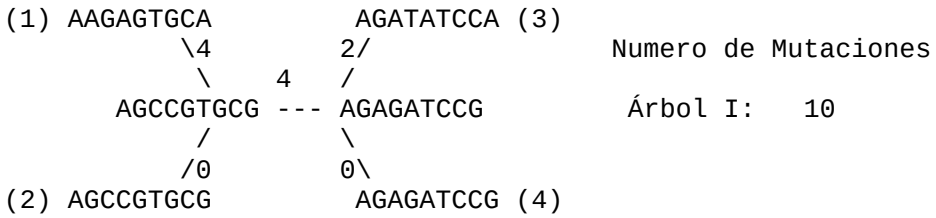
Donde $L_{(\tau)}$ es la longitud del árbol, **B** es el numero de ramas, **N** es el número de caracteres, **k'** y **k''** son los dos nodos incidentes para cualquier rama **k**, $x_{k'j}$, $x_{k''j}$ representa cualquier elemento de la matriz de datos de entrada o las asignaciones carácter-estado optimas hechas a los nodos internos y **diff(y , z)** es una función especificando el costo de una transformación de un estado **y** a un estado **z** a lo largo de cualquier rama. El coeficiente w_j asigna un peso a cada carácter. Note además que diff(y,z) no necesariamente es igual a diff(z,y). Para métodos que producen arboles sin raíz diff(z,y)=diff(y,z).

¿Cómo Funciona MP?

Consideremos el siguiente ejemplo:

Secuencias	1	2	3	4	5	6	7	8	9
1	A	A	G	A	G	T	G	C	A
2	A	G	C	C	G	T	G	C	G
3	A	G	A	T	A	T	C	C	A
4	A	G	A	G	A	T	C	C	G

Para estas cuatro OTUs hay tres posibles arboles sin raíz. Los arboles son analizados en busca de secuencias ancestrales y contando el numero de mutaciones requeridas para explicar los respectivos arboles.



El Árbol I tiene la topología con el menor número de mutaciones y así es el árbol más parsimonioso. El anterior análisis se basa en todos los sitios de la secuencia alineada. Sin embargo un número de sitios son no informativos y, por lo tanto, no tienen que ser incluidos en el análisis. Cuando solo sitios informativos son incluidos un numero mucho menor de sitios puede ser analizados, lo cual significa en el caso de grandes conjuntos de datos una ganancia considerable en tiempo de computación.

Sitios Informativos: un sitio es informativo solo cuando hay al menos dos tipos diferentes de nucleótidos en el sitio, cada uno de los cuales está representado en al menos dos de las secuencias bajo estudio. Para ilustrar la diferencia entre sitios informativos y no informativos, miremos las siguientes secuencias hipotéticas:

	Sitio								
Secuencia	1	2	3	4	5	6	7	8	9
1	A	A	G	A	G	T	G	C	A
2	A	G	C	C	G	T	G	C	G
3	A	G	A	T	A	T	C	C	A
4	A	G	A	G	A	T	C	C	G
					*		*		*

Hay tres posibles arboles sin raíz para la cuatro OTUs (árbol I, árbol II y árbol III, ver a continuación). El sitio 1 es no informativo porque todas las secuencias en este sitio tienen A, de modo que ningún cambio es requerido en cualquiera de los tres arboles posibles. En el sitio 2, la secuencia 1 tiene una A mientras todas las otras secuencias tienen una G y una asunción simple es que el nucleótido ha cambiado de G a A en el linaje que conducía a la secuencia 1. Así este sitio también es no informativo, porque cada uno de los tres posibles arboles requiere un cambio. Como se muestra, para el sitio 3 cada uno de los

tres posibles arboles requieren 2 cambios y así este es también no informativo. Como vemos para el sitio 4 cada uno de los tres arboles requiere 3 cambios y así este sitio es también no informativo. Para el sitio 5, el árbol I requiere solo 1 cambio, mientras que los arboles II y III requieren 2 cambios cada uno, por lo tanto este sitio es informativo. En el ejemplo los sitios informativos 5,7 y 9 están marcados con *.

A continuación cuatro secuencias y sus correspondientes tres posibles arboles hechos solo con sitios informativos:

```
              Sitio
              579
Secuencia
       1      GGA
       2      GGG
       3      ACA
       4      ACG
              ***
```

```
(1)  GGA          ACA (3)
       \1         1/                Número de mutaciones
        \    2    /
         GGG --- ACG                  Árbol I:   4
        /          \
       /0          0\
(2)  GGG          ACG (4)
```

```
(1)  GGA          GGG (2)
       \1         1/
        \    1    /
         GCA --- GCG                  Árbol II:  5
        /          \
       /1          1\
(3)  ACA          ACG (4)
```

```
(1)  GGA          GGG (2)
       \2         1/
        \    0    /
         GCG --- GCG                  Árbol III: 6
        /          \
       /1          2\
(4)  ACG          ACA (3)
```

Para inferir el árbol de máxima parsimonia, para cada posible árbol calculamos el mínimo número de sustituciones en cada sitio informativo. En el anterior ejemplo para los sitios 5, 7 y 9, el árbol I requiere en total 4 cambios, el árbol II requiere 5 cambios y el árbol III requiere 6 cambios. En el paso final, sumamos el número de cambios sobre todos los sitios informativos para cada árbol y escogemos el árbol asociado con el más pequeño número de sustituciones. En este caso el árbol I es escogido porque requiere el más pequeño número de cambios (4) en los sitios informativos.

MAXIMUM LIKELIHOOD

MAXIMUM LIKELIHOOD - ML (Máxima Verosimilitud), o método de máxima probabilidad computa la probabilidad de obtener los datos D (las secuencias alineadas observadas) dada una hipótesis H definida (el árbol y el modelo de evolución). Esto es:

$$P\ (D/H)$$

Los métodos de máxima verosimilitud evalúan que tan bien un modelo de proceso evolutivo explica los datos observados.

El ejemplo de la moneda: Estimación de la máxima verosimilitud de la probabilidad de que sea cara (H) una moneda que es lanzada n veces. Sí los lanzamientos son todos independientes, y todos tienen la misma probabilidad desconocida p que sea cara, entonces la secuencia de observación de lanzamientos es:

HHTTHTHHTTT

Podemos calcular la máxima verosimilitud de estos datos así:

$$L = P(D/p) = pp(1-p)(1-p)p(1-p)pp(1-p)(1-p)(1-p) = p^5(1-p)^6$$

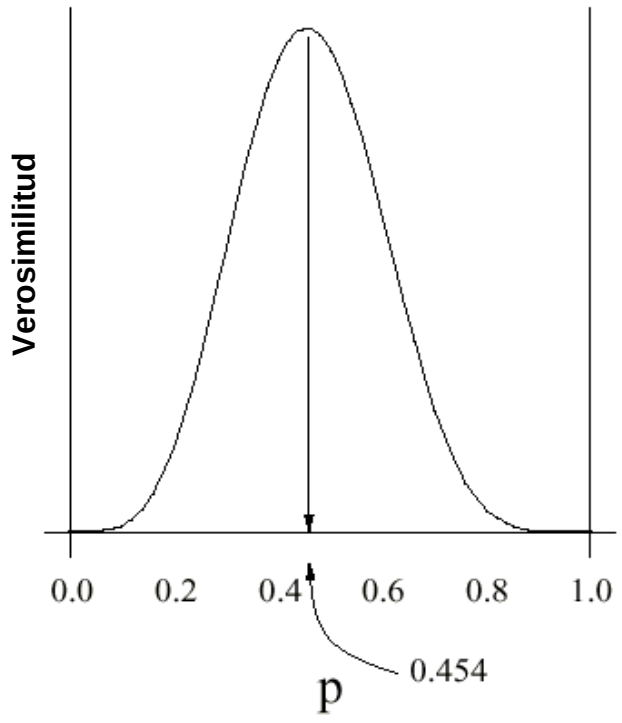


Fig. 2.9. Probabilidad que sea cara una moneda que es lanzada n veces.

Dibujando L (Verosimilitud) contra p Fig. 2.9, se observa las probabilidades de los mismos datos D para diferentes valores de p. Así la máxima verosimilitud o la máxima probabilidad a observar en la anterior secuencia de eventos es p=0.4545, estos es:

$$\frac{5}{11} \rightarrow \left(\frac{Caras}{Caras+Sellos}\right)$$

Esto puede ser verificado calculando la derivada de L respecto a p:

$$\frac{dL}{dp} = 5p^4(1-p)^6 - 6p^5(1-p)^5$$

Igualándola a cero, y resolviéndola:

$$\frac{dL}{dp} = p^4(1-p)^5[5(1-p) - 6p] = 0 \rightarrow p = 5/11$$

Más fácilmente, las verosimilitudes son a menudo maximizadas maximizando sus logaritmos:

$$\ln L = 5\ln p + 6\ln(1-p)$$

y su derivada es:

$$\frac{dln}{dp} = \frac{5}{p} - \frac{6}{1-p} = 0 \rightarrow p = \frac{5}{11}$$

La verosimilitud de una secuencia: Supongamos que tenemos los siguientes elementos,

Datos: Una secuencia de 10 nucleótidos de longitud, AAAAAAAATG

Modelo: Jukes-Cantor $\rightarrow f_{(A,C,G,T)} = \frac{1}{4}$

Modelo: Modelo$_1$ $\rightarrow f_{(A,C,G,T)} = \frac{1}{2}, \frac{1}{5}, \frac{1}{5}, \frac{1}{10}$

$$L_{JC} = \left(\frac{1}{4}\right)^8 . \left(\frac{1}{4}\right)^0 . \left(\frac{1}{4}\right) . \left(\frac{1}{4}\right) = 9.53x10^{-7}$$

$$L_{M1} = \left(\frac{1}{2}\right)^8 . \left(\frac{1}{5}\right)^0 . \left(\frac{1}{5}\right) . \left(\frac{1}{10}\right) = 7.81x10^{-5}$$

L_{M1} es casi 100 veces mayor que el modelo L_{JC}. Por lo que el modelo Jukes-Cantor no es el mejor modelo para explicar estos datos.

Dado que la verosimilitud toma la forma de:

$$\prod_{i=1}^{n} Xi$$

donde 0 ≤ Xi ≤ 1, y generalmente n es grande, es conveniente reportar el resultado de máxima verosimilitud como $\ln L$ o $\log_{10} L$.

$$\text{lnL(JC)} = -13.8636 \text{ ; lnL(M1)} = -9.4575$$

donde el más positivo (el menor valor negativo de lnL) es la mejor verosimilitud.

2.4. DISEÑO DE ALGORITMO PARALELOS

Hoy en día muchos problemas de programación tradicionales han sido paralelizados o por lo menos migrados a plataformas de alto rendimiento. La solución típica aplicada a la mayoría de problemas de programación es paralelizar la mejor solución secuencial, pero puede resultar que no sea la mejor solución paralela. La metodología en que se basa el diseño de algoritmos paralelos, según Ian Foster, comprende cuatro etapas: Particionamiento, Comunicación, Agrupación y Asignación.

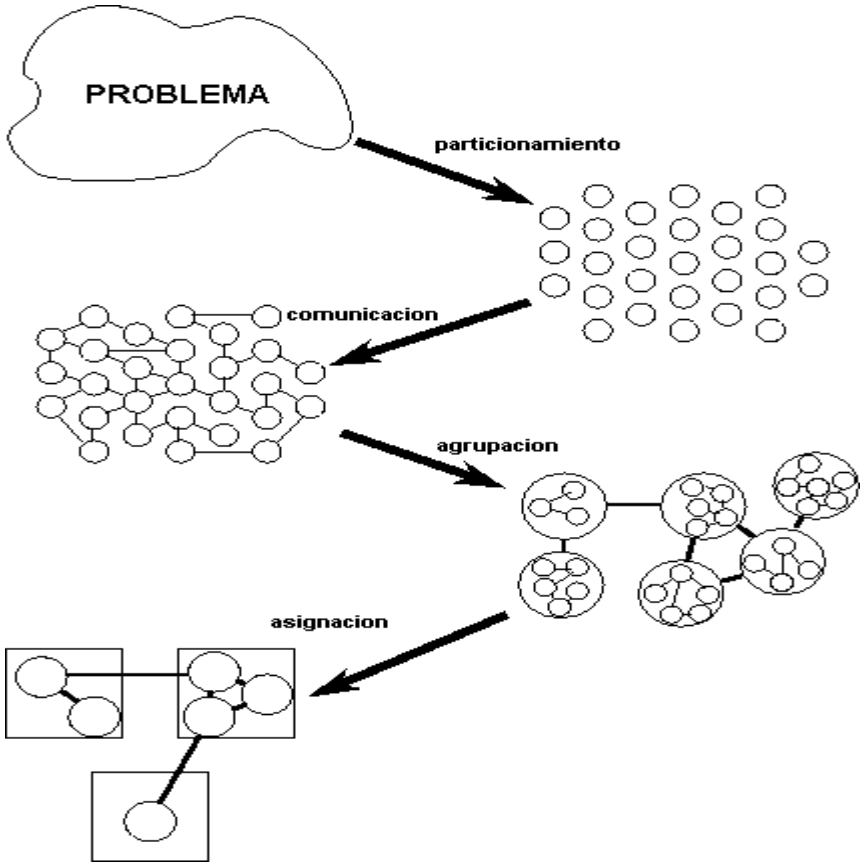


Fig. 2.10. Etapas del Diseño de Algoritmos Paralelos (Ian Foster, Designing and Building Parallel Programs)

La siguiente tabla describe brevemente las cuatro etapas del diseño de algoritmos paralelos.

Tabla 2.7. Etapas del Diseño de Algoritmos Paralelos.

Etapa	Descripción
Particionamiento	Los datos y el computo a realizarse sobre estos se descomponen en pequeñas tareas. En esta etapa la atención se centra en encontrar oportunidades de paralelización, elementos como el numero de procesadores en la maquina son ignorados.
Comunicación	La comunicación requerida para coordinar la ejecución de las tareas es definida, así como las estructuras y algoritmos apropiados de comunicación.
Agrupación	En esta etapa se evalúan las tareas y estructuras de comunicación con respecto al costo de implementación y los requisitos de funcionamiento. Si es necesario las tareas son combinadas en tareas más grandes para mejorar el funcionamiento o reducir costos de implementación.
Asignación	Cada tarea es asignada a un procesador de tal forma que intenta satisfacer las metas de maximizar la utilización del procesador y minimizar los costos de comunicación. La asignación puede ser estática o determinada en tiempo de ejecución por algoritmos de balanceo de carga.

Particionamiento

La etapa de particionamiento se entiende como buscar las oportunidades de ejecución paralela. El enfoque esta en definir una gran cantidad de tareas pequeñas para hacer una *granularidad fina*. Cuando la granularidad es fina es más fácil apilar las pequeñas tareas. Una descomposición de grano fino provee una excelente flexibilidad en términos de aumentar el potencial de paralelización. En las siguientes fases, de acuerdo a la evaluación de requerimientos de comunicación, la arquitectura o software de desarrollo, se descartarán algunas oportunidades de paralelización.

Una buena paralelización divide en pequeñas piezas: se realiza división de instrucciones de máquina o división de los datos. Cuando se realiza una descomposición de los datos asociados con el problema se denomina *descomposición de dominio*. Otra técnica alternativa, primero descomponer el número de tareas, y después descomponer los datos se llama *descomposición funcional*.

- *Descomposición de Dominio*

Con la descomposición de datos se da una aproximación a un problema de particionamiento, lo primero que se busca es la descomposición de los datos asociados al problema. Si es posible se busca dividir en pequeñas piezas del mismo tamaño. Después que la partición está hecha, se asocia cada operación con los datos con los cuales va a operar. Esta partición produce un número de tareas, cada una restringida a los datos sobre los cuales va a operar. Una tarea puede requerir datos de diferentes tareas. En ese caso se necesita un canal para comunicar las diferentes tareas. Este requerimiento se analiza en la siguiente etapa.

Los datos pueden ser distribuidos en la entrada del problema, en la salida o mientras se realiza el proceso. Se puede realizar diferentes tipos de particiones, basado en diferentes estructuras de datos. Se sugiere que se empiece por la estructura de datos principal, o la que tenga más acceso a lo largo del programa.

La figura 14 muestra la descomposición de un problema que tiene una malla de 3 dimensiones. Las instrucciones son ejecutadas en cada punto de la malla repetidamente. La descomposición en el eje x, y o z se puede realizar de la forma más fácil. La descomposición más agresiva en este caso sería, en cada punto de la malla. Cada tarea mantendría en un estado varios valores asociados al punto de la malla y es responsable por el cómputo que sea necesario para actualizar su estado.

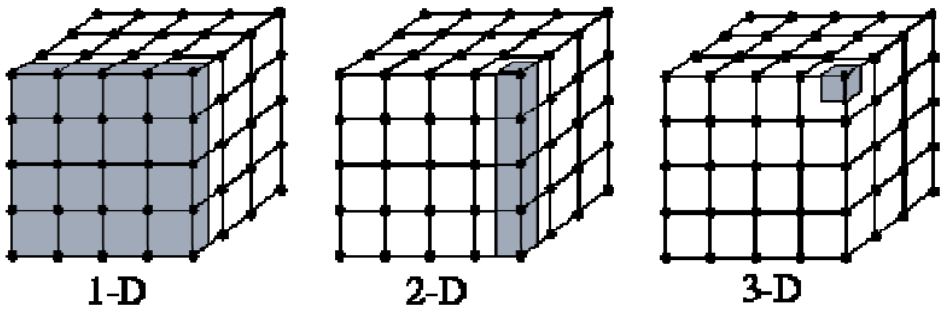


Fig. 2.11. Descomposición de dominio que involucra una malla de tres dimensiones.

- *Descomposición Funcional*

La descomposición funcional ofrece una alternativa diferente y complementaria. En esta aproximación la atención se centra en la computación que será ejecutada sin pensar en los datos que se utilizarán. Después de realizar la división en diferentes tareas, se procede a revisar los datos que se necesitarán para realizarla. Los requerimientos de datos pueden ser diferentes para cada tarea, en ese caso la partición ha sido completa. De lo contrario aparecen solapamientos de datos, en este caso será necesaria una cantidad de comunicación considerable. Esta es una razón por la cual se prefiere realizar una descomposición de datos.

Un ejemplo de cuando la descomposición funcional puede ser la más apropiada, es la exploración en un árbol para la búsqueda del nodo “soluciones”. La descomposición del dominio seria obvia. Sin embargo una descomposición funcional de grano fino puede plantearse: inicialmente una tarea simple se crea en el nodo raíz del árbol, la tarea evalúa

ese nodo, y si no es el nodo buscado crearía una nueva tarea (subárbol). Una nueva tarea sería creada cada vez que se expanda la búsqueda en el árbol.

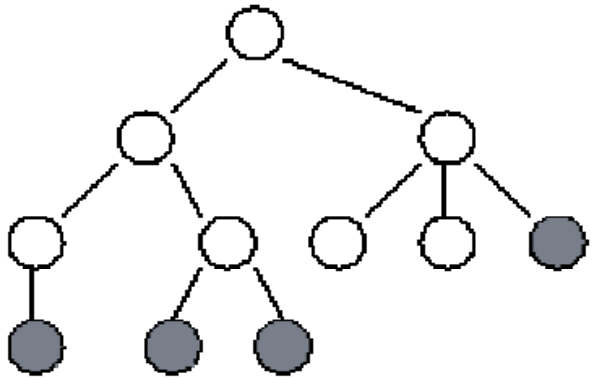


Fig. 2.12. Estructura de Tarea para un ejemplo de búsqueda.

Comunicación

Las tareas generadas en la partición son pensadas para ejecutarse concurrentemente, pero en general no son independientes. Las tareas generalmente requieren datos entre ellas. Los datos deben ser transmitidos de una tarea a otra para que pueda proseguir con su cómputo. Este flujo de información debe ser especificado en la etapa de comunicación.

Entonces surge la necesidad de conceptuar comunicación entre tareas, como un canal que las une, en el cual una tarea envía mensajes y otra la recibe. Entonces la comunicación asociada con un algoritmo puede ser especificada en 2 partes: La primera define un canal que une, directa o indirectamente, tareas que requieren datos (consumidor), con las tareas que poseen esos datos (productor). En la segunda se especifica los mensajes que serán enviados y recibidos por ese canal.

La comunicación se puede clasificar en local y global.

- Comunicación Local

Una estructura de comunicación local se obtiene cuando una operación necesita datos de un pequeño número de tareas Cuando este tipo de comunicación ocurre se debe crear un canal entre el consumidor y productor de tareas, e introducir operaciones de recepción y envío entre las tareas. Por ejemplo, los requerimientos de comunicación asociados con un método numérico simple: El método de diferencias finitas de Jacobbi. La siguiente expresión se usa para actualizar cada uno de los puntos X_{ij} en una malla X.

$$X_{i.j}^{t+1} = \frac{4X_{i.j}^{t} + X_{i.-1.j}^{t} + X_{i+1.j}^{t} + X_{ij-1}^{t} X_{ij+1}^{t}}{8}$$

Esta actualización es aplicada repetidamente cuando $t=1,2,3, .. , n$ .La notación $X_{i.j}^{t}$ simboliza el valor del punto en la malla $X_{i,j}$ en el paso de tiempo t.

- Comunicación Global

Una operación de comunicación global es aquella en la que varias tareas tienen que participar. Cuando cada operación es implementada, es difícil distinguir cada par de consumidores y productores. Por ejemplo considere que se necesita recoger los resultados de N operaciones realizadas en N tareas usando el operador de la suma.

$$S=\sum_{i=0}^{n-1} X_i$$

Se asume una tarea líder que requiere el resultado de S. Desde el punto de vista local, se deben crear canales independientes en donde cada tarea 0,1,2,3,…,n envía su *Xi* a la tarea líder que debe acumular cada uno de los recibos en la variable S, como se muestra en la figura 16.

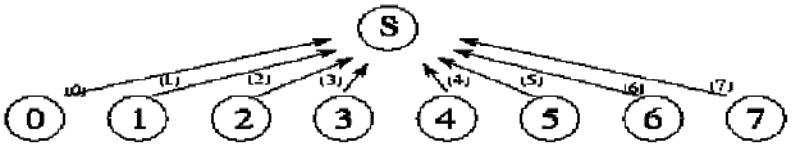


Fig. 2.13. Algoritmo centralizado de suma que usa una tarea líder S para sumar N números distribuidos a lo largo de n tareas.

Agrupación

En las primeras dos etapas del proceso de diseño, se realiza la partición de la computación ejecutada en pequeñas tareas y se introduce la computación requerida por esas tareas. El algoritmo resultante es abstracto debido a que no está orientado a una máquina paralela en especial. Este factor puede hacerlo ineficiente.

En esta tercera etapa, la agrupación, se mueve de lo abstracto a lo concreto. Se revisa si el algoritmo que se obtuvo en las 2 fases anteriores se adecúa a la maquina paralela donde se piensa implementar. El número de tareas agrupadas en esta fase, que deben tender a la reducción, deben ser más grande que el número de procesadores a utilizar. Alternativamente, durante la etapa de agrupación se puede reducir el número de tareas al número de procesadores a utilizar.

Tres factores influyen a la hora de agrupar: granularidad, flexibilidad y costos del software.

- *Incrementar la Granularidad*

Una parte crítica que influencia el rendimiento de un algoritmo paralelo son los costos de comunicación. En la mayoría de maquinas paralelas, se tiene que parar la computación, para recibir y enviar mensajes.

Para resolver este problema se puede tomar dos caminos:

- Mejorar el rendimiento enviando menos datos
- Disminuir el número de envíos realizados pero con la misma cantidad de datos

- *Preservar la Flexibilidad*

La habilidad de crear tareas variables es crítica si se quiere que el programa sea escalable y portable. Esta flexibilidad es útil para adecuar el código para un computador en particular. Crear más tareas que procesadores permite una gran variedad de opciones en la etapa de asignación, permitiendo un mejor balanceo de carga.

- *Reduciendo costos en la Ingeniería del Software*

Una consideración adicional, cuando existen códigos secuenciales, es el costo relativo asociado con las diferentes técnicas de partición. Desde este punto de vista se puede escoger no realizar cambios significativos. Por ejemplo, si el código opera una malla multidimensional, puede ser ventajoso preservar las particiones alrededor de una dimensión, si este posee rutinas que no puedan ser cambiadas en el programa paralelo.

Asignación

La última etapa del proceso de diseño es la asignación, donde se especifica donde se va a ejecutar cada tarea. En esta clase de computadores, se requiere que el grupo de tareas y requerimientos de comunicación se encuentren bien especificados en el algoritmo paralelo; sistemas operativos o mecanismos de hardware, puede ayudar a asignar las tareas ejecutables en los procesadores disponibles. Desafortunadamente, los mecanismos de mapeo deben ser realizados para poder ser ejecutados en computadores paralelos escalables. En general, la asignación es un problema difícil que debe ser asignado explícitamente cuando se diseña algoritmos paralelos.

La meta al desarrollar algoritmos paralelos es minimizar el tiempo total de ejecución. Se pueden usar dos estrategias para lograrlo:

✓ Asignar tareas que se pueden ejecutar concurrentemente en diferentes procesadores para evitar la concurrencia.
✓ Asignar tareas que se comuniquen frecuentemente en el mismo procesador para incrementar la comunicación entre procesadores.

Estas estrategias, algunas veces causan conflicto, produciendo un desbalanceo de carga. Además las limitaciones de recursos pueden restringir el número de tares asignadas a un procesador.

Muchos algoritmos desarrollados que usan descomposición de dominio usan la técnica de fijar tareas de igual tamaño, comunicación local estructurada y global. En estos casos una asignación eficiente es sencilla. Un algoritmo más complejo en el cual el trabajo por tarea y/o comunicación no posean una estructura, hará que la agrupación y descomposición de dominio no sea tan obvia para el programador. Es allí donde se utiliza algoritmos de *balanceo de carga* que buscan identificar estrategias eficientes de agrupación y asignación, usualmente técnicas heurísticas. El tiempo requerido para ejecutar estos algoritmos debe ser “pesado” con el fin de ganar en tiempo de ejecución.

Los problemas más complejos son aquellos en los cuales el número de tareas o la cantidad de comunicación o computación cambia dinámicamente durante el tiempo de ejecución. En ese caso se puede usar *balanceo de carga dinámico*, con la cual un algoritmo es ejecutado periódicamente para determinar la nueva aglomeración y asignación.

Los algoritmos basados en descomposición funcional producen frecuentemente tareas que coordinan a otras tareas al inicio y final de la ejecución. En ese caso se pueden usar algoritmos de *programación de tareas*, que asignan tareas a un procesador que se encuentre ocioso.

3. UPGMA: RECONSTRUCCION DE ARBOLES FILOGENETICOS

3.1. METODOLOGIA DE TRABAJO

La metodología elegida para este proyecto, luego de estudiadas las características del mismo y analizados los diferentes modelos de ciclo de vida(lineal secuencial o modelo en cascada, construcción de prototipos o prototipado evolutivo, modelo en espiral o modelos evolutivos, el modelo de proceso DRA, basado en componentes u orientado a objetos; según Roger S. Pressman[5]) es el Prototipado Evolutivo.

El modelo elegido, Prototipado Evolutivo, se sustenta para este proyecto luego de observar las necesidades y características que en sí posee, y que a continuación se enuncian:

- El proyecto trabaja con poca identificación de los requerimientos y de las especificaciones.
- El proyecto trabaja con un algoritmo del cual no se tiene certeza de su eficacia y pudiera no ser adecuado.
- El proyecto permite modificaciones a las especificaciones a medio camino

El Prototipado Evolutivo ofrece el mejor enfoque al desarrollo del proyecto, gracias a sus ventajas que se ajustan a las características y solucionan sus necesidades; dichas ventajas son:

- El modelo no exige una fuerte planificación.
- El desarrollo del prototipo se basa en la realimentación que recibe, para luego agregar nuevas funcionalidades hasta cumplir los objetivos propuestos.
- El concepto del sistema se desarrolla a medida que avanza el proyecto.
- Se pueden realizar cambios en etapas preliminares y así emitir varios prototipos evaluables durante el desarrollo, obteniéndose conjuntamente una metodología para el proceso de evaluación del sistema.
- Genera signos visibles de progreso.

[5] Pressman, Roger. Ingeniería del Software. Un Enfoque Práctico. Quinta Edición. Mc Graw Hill. España. 2002

- Genera un intercambio de conocimientos y autocrítica al sistema, lo que propicia que se ejecuten muchas pruebas antes de liberar una nueva versión así como mejoras rápidas a problemas que puedan surgir durante su uso.

Luego el paradigma de construcción de prototipos comienza con la recolección de requisitos, búsqueda y definición de los objetivos globales del sistema, identificación de los requisitos conocidos y las áreas donde se hace necesario una mayor definición. Entonces aparece un "diseño rápido" que conlleva a la construcción de un prototipo, el prototipo lo evalúa el cliente/usuario y se utiliza para refinar los requisitos del software a desarrollar. Finalmente se produce un proceso iterativo en el que el prototipo es puesto a punto para satisfacer las necesidades del cliente permitiendo al mismo tiempo que el desarrollador comprenda mejor lo que necesita hacer.[6]

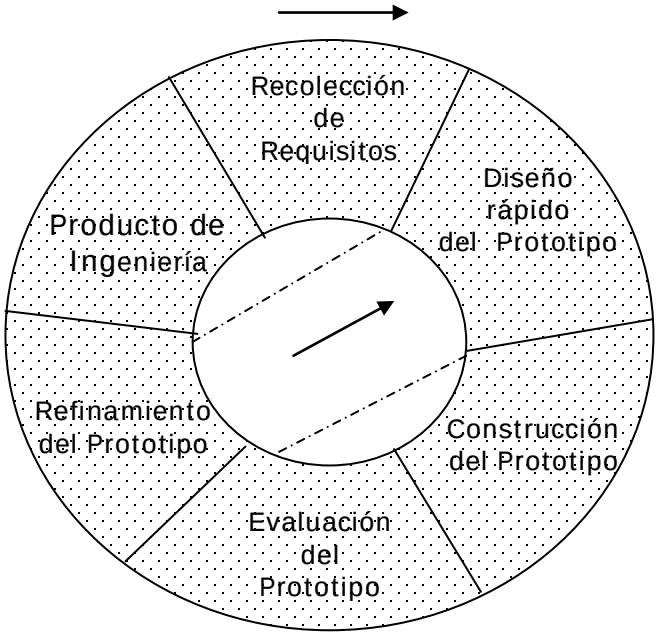


Fig. 3.1. Modelo de Prototipado Evolutivo

Fuente: Pressman, Roger. Ingeniería del Software. Un Enfoque Práctico.

En el proyecto se distinguen dos etapas de prototipado:

La primera etapa o secuencial, de acuerdo a los objetivos planteados, analiza, diseña e implementa una solución de tipo secuencial, la cual permite recibir un archivo de secuencias(en formato FASTA[7]) construido a partir de la búsqueda en una base de datos como NCBI, y luego de construir la matriz de distancias y aplicar el algoritmo UPGMA se genera un archivo en con la filogenia completa en formato Newick[8], que nos permitirá

[6] Pressman, Roger. Ingeniería del Software. Un Enfoque Práctico. Quinta Edición. Mc Graw Hill. España. 2002

[7] http://www.ncbi.nlm.nih.gov/blast/fasta.shtml

[8] http://evolution.genetics.washington.edu/phylip/newicktree.html

entender comportamientos similares a nivel biológico entre diferentes especies, poblaciones u otras unidades de taxonomía.

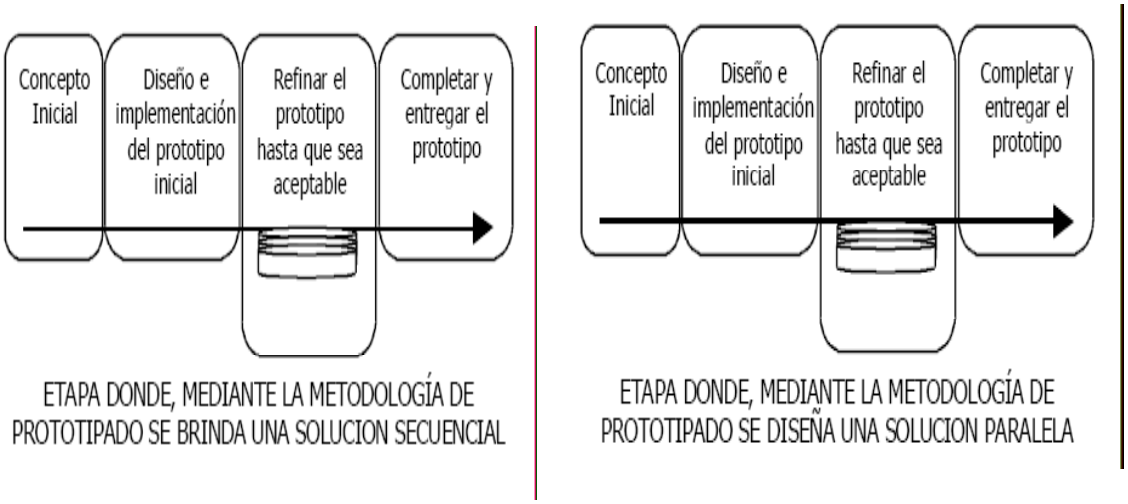


Fig. 3.2. Etapas de la metodología de prototipado evolutivo

La segunda etapa o paralela, aprovecha la solución secuencial obtenida en la primera etapa, la analiza, le define requisitos, diseña la solución paralela del algoritmo UPGMA, y la implementa, dando como resultado un producto que retorna la misma la solución secuencial, pero que en teoría debe trabajar de forma más rápida y eficiente.

3.2. PLAN DE TRABAJO

De acuerdo a la metodología planteada, se definieron tres fases como componentes del plan de trabajo. Las mismas son:

- *Fase Inicial:*

  - ✓ *Recopilación de Información:* Esta etapa comprende la recopilación del material bibliográfico necesario para dar soporte al desarrollo del proyecto.
  - ✓ *Entrenamiento Previo:* En esta etapa se lleva a cabo el estudio y la familiarización de la teoría filogenética, así como conocimientos generales en el área de la bioinformática, y por último, conocer las técnicas de programación empleadas en el lenguaje de programación Python. Participando también de la pasantía en Ciencias Computacionales en CeCalcULA, asistiendo al III Congreso de Bioinformática de la ULA y al Taller de Herramientas Bioinformáticas de la ULA.

- ✓ *Análisis y Especificación:* En esta etapa se definen los recursos necesarios para la elaboración del proyecto, además realiza la recopilación y análisis de los requerimientos del sistema.

- *Fase Iterativa:* Esta fase comienza con la realización del prototipo inicial, tomando como base las especificaciones de los requerimientos de la fase inicial.

  - ✓ *Diseño:* realiza la identificación de los parámetros necesarios para implementar el algoritmo UPGMA. Posteriormente se implementa dicho algoritmo en forma secuencial para que finalmente se desarrolle en paralelo.
  - ✓ *Desarrollo:* realiza la implementación del prototipo definido para el ciclo actual, basándose en el diseño definido en la etapa anterior.
  - ✓ *Plan de pruebas:* realizan las pruebas necesarias para obtener indicadores cuantitativos y cualitativos del funcionamiento del sistema.
  - ✓ *Análisis de resultados:* realiza una evaluación de los resultados obtenidos de la etapa anterior, para decidir si se avanza a la siguiente fase, o se vuelve a la fase iterativa.

- *Fase de evaluación:* Se efectúa un estudio de rendimiento (profiling) de la implementación del algoritmo en paralelo teniendo como base el código desarrollo en este trabajo de grado en forma secuencial. Dentro de esta comparación se evalúan aspectos como rendimiento, eficiencia, esfuerzo computacional, y otros a fin de emitir una conclusión respecto a la conveniencia de paralelizar el algoritmo UPGMA.

3.3. DESARROLLO SECUENCIAL

Para lograr plasmar UPGMA[9] - Unweighted Pair Group Method with Arithmetic mean de forma tal que se convirtiera en pyUPGMA secuencial fue necesario un análisis exhaustivo del funcionamiento del algoritmo, lo que nos permitió encontrar cuatro etapas: Entrada, Inicialización, Iteración y Salida.

El siguiente esquema nos ilustra las etapas encontradas en el desarrollo secuencial:

[9] Sokal, R. R. and C. D. Michener. 1958. A statistical method for evaluating systematic relationships. University of Kansas Science Bulletin 38: 1409-1438

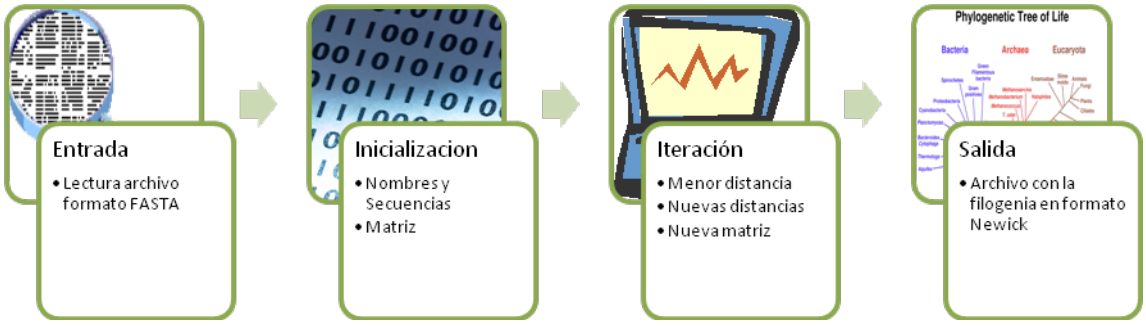


Fig. 3.3. Etapas en el Desarrollo Secuencial

3.3.1. Entrada

Los datos que procesa el algoritmo son las distancias entre las secuencias (alineadas) y que luego se traducen en forma de matriz para que sea entendible por el programa.

Fig. 3.4. Datos de entrada: La distancia entre las secuencias W y X es tres, igual al número de sitios diferentes entre las secuencias W y X.

Desde el punto de vista del software esta etapa se plasma en las siguientes funciones creadas para tal fin:

Tabla 3.1. Funciones de la Etapa de Entrada

Función	Parametros	Objetivo
leamultif(archivo)	archivo: nombre del archivo que contiene los datos de las secuencias en formato FASTA.	Carga los datos contenidos en el archivo con formato FASTA
creanom(temp)	temp: lista de nombres y secuencias sacadas de "archivo"	Crea un listado solo con los nombres de las secuencias
creasec(temp)	temp: lista de nombres y secuencias sacadas de "archivo"	Crea un listado solo con las secuencias

3.3.2. Inicialización

Luego de obtener los datos de entrada procedemos a organizar los datos de tal forma que podamos construir la matriz de distancias, además claro el árbol filogenético.

1. Inicializar n clúster con un OTU(secuencia en nuestro caso) por cada clúster.
2. Colocar el tamaño de cada clúster |Ci|=1.
3. Colocar una hoja por cada clúster Ci en el árbol ultramétrico T.
4. Asignar D=M, donde D es la matriz de distancias que ira cambiando a lo largo del algoritmo.

Las funciones que nos permitirán llevar a cabo esta inicialización desde el programa como tal son:

Tabla 3.2. Funciones de la Etapa de Inicialización

Función	Parametros	Objetivo
matriz(secuens)	secuens: una lista con las secuencias de nucleótidos o aminoácidos	Construye la matriz de distancias en base a las secuencias
compfasta(s1,s2)	s1,s2: secuencias de aminoácidos o nucleótidos	Determina el numero de diferencias entre las secuencias s1 y s2

3.3.3. Iteración

Esta etapa tiene la mitad del desarrollo del algoritmo por lo que es importante conocer los pasos que se siguen.

1. Encontrar los Ci y Cj con la menor distancia dij.
2. Crear un Nuevo cluster C(ij) que tenga | C(ij)|=|Ci|+|Cj| miembros
3. Conectar en el arbol T a Ci y a Cj con el nuevo nodo C(ij) y darle a las nuevas ramas que los conectan una distancia de dij/2.
4. Determinar las nuevas distancias del cluster C(ij) a todos los demas clusters como un promedio de las distancias de sus componentes:

   d(ij),k=[(dik)+(djk)]/2
5. Borrar las columnas en D correspondientes a los cluster Ci y Cj y agregar una nueva columna para el cluster C(ij).
6. Volver al paso 1 hasta que solo quede un cluster.

Las funciones que nos conduciran a iterar el algoritmo como deseamos desde el software son:

Tabla 3.3. Funciones de la Etapa de Iteración

Función	Parametros	Objetivo
matriz(secuens)	secuens: una lista con las secuencias de nucleótidos o aminoácidos	Construye la matriz de distancias en base a las secuencias
compfasta(s1,s2)	s1,s2: secuencias de aminoácidos o nucleótidos	Determina el numero de diferencias entre las secuencias s1 y s2
upgma(d,nombres)	d: matriz de distancias nombres: nombres de las secuencias	Construye una nueva matriz de distancias acorde al algoritmo UPGMA
repite(names,grilla)	names: nombres de las secuencias grilla: matriz que va cambiando con la iteración	Repetir el algoritmo UPGMA tantas veces como filas tenga la matriz de distancias inicial

3.3.4. Salida

Las funciones de esta etapa para terminar el desarrollo secuencial son:

Tabla 3.4. Funciones de la Etapa de Salida

Función	Parámetros	Objetivo
fnewick(lista)	lista: contiene en formato de lista toda la filogenia obtenida durante la iteración	Modifica la filogenia obtenida acorde al formato Newick
salida(archivo,filo_newick)	archivo: es el nombre del archivo en formato FASTA inicial filo_newick: es la filogenia completa convertida en formato newick	Crea un archivo de texto con la filogenia completa

La salida que se obtiene después de ejecutar el algoritmo en su totalidad es un árbol T ultramétrico basado en la matriz de distancias M. Para poder reproducir gráficamente el árbol se almacena la salida en un archivo en formato Newick.

4. DESARROLLO DE PYUPGMA

4.1. RECOLECCION Y ANALISIS DE REQUISITOS

4.1.1. Entendimiento del Problema

La primera fase del proyecto fue diseñar e implementar el algoritmo UPGMA de forma secuencial, finalizada esta etapa se tiene conocimiento del problema al que se va a enfrentar de una manera mucho más clara y amplia. A continuación se describen las características del programa secuencial desarrollado en este proyecto:

- **Características**

pyUPGMA Secuencial determina la filogenia para secuencias de nucleótidos o aminoácidos, basado en el método Unweighted Pair Group Method with Mean Average presentado por Sokal & Michener en “A statistical method for evaluating systematic relationships” - University of Kansas Science Bulletin 38: 1409-1438; como se ha descrito en los capítulos anteriores.

- **Formulación**

El algoritmo UPGMA como se indicó en el capítulo 2 se basa en el clustering (agrupamiento) de secuencias y en las distancias (como tipo de dato) entre secuencias, para la reconstrucción de la filogenia.

Luego de construir la matriz de distancias a partir de un archivo de datos en formato FASTA, comienza por calcular la menor distancia entre secuencias, con las dos secuencias menos distantes se crea un nuevo grupo (clúster) que está formado por la distancia promedio entre las secuencias que forman el nuevo clúster y las demás secuencias, este grupo se agrega a la matriz y luego se eliminan de la matriz las secuencias que crearon el nuevo grupo. Creando una nueva matriz de distancias. Una vez más se busca la menor distancia con los nuevos datos, y se repite el proceso hasta obtener un único grupo.

En la medida en que se realiza el proceso iterativo de UPGMA se van anotando los valores de las distancias entre las secuencias menos distantes que permitirán reconstruir el árbol filogenético, a estos datos se les da un formato NEWICK que es el más utilizado para poder reconstruir el árbol gráficamente en terceras aplicaciones.

- **Datos de Entrada**

El formato de datos de entrada es en Fasta. Que comienza con una linea de descripcion de la secuencia. La línea de descripción se distingue de los datos de la secuencia por que comienza con el simbol mayor-que (“>”) en la primera columna, luego aparece un nombre o identificador unico para la secuencia, y el resto de la línea es información adicional.

Un ejemplo de una secuencia en formato fasta:

```
>gi|5524211|gb|AAD44166.1| cytochrome b [Elephas maximus maximus]
LCLYTHIGRNIYYGSYLYSETWNTGIMLLLITMATAFMGYVLPWGQMSFWGATVITNLFSAIPYIGTNLV
EWIWGGFSVDKATLNRFFAFHFILPFTMVALAGVHLTFLHETGSNNPLGLTSDSDKIPFHPYYTIKDFLG
LLILILLLLLLALLSPDMLGDPDNHMPADPLNTPLHIKPEWYFLFAYAILRSVPNKLGGVLALFLSIVIL
```

Para las pruebas se usaran datos de entrada con varias secuencias que tendrán la siguiente forma:

>SEQUENCE_1

MTEITAAMVKELRESTGAGMMDCKNALSETNGDFDKAVQLLREKGLGKAAKKADRLAAE
GLVSVKVSDDFTIAAMRPSYLSYEDLDMTFVENEYKALVAELEKENEERRRLKDPNKPEH

>SEQUENCE_2

SATVSEINSETDFVAKNDQFIALTKDTTAHIQSNSLQSVEELHSSTINGVKFEEYLKSQIATI
GENLVVRRFATLKAGANGVVNGYIHTNGRVGVVIAAACDSAEVASKSRDLLRQICM

- **Salida del Programa**

El programa produce un archivo cuya extensión será ".newick" alusiva al formato que tendrán los datos de salida. El nombre del archivo estará conformado por un encabezado "Filo_" para que nos permita fácilmente identificarlo del archivo de entrada. Quedando de la siguiente forma:

Filo_HIV1.fasta.newick

El contenido de la salida será como la siguiente:

(U68502,((U68508,((U68506,U68504):4.0,(U68507,U68505):5.5):6.5):13.625,
(U68503,(U68501,((U68496,U68497):4.5,(U68498,(U68499,U68500):5.5):8.2
5):11.875):13.0625):15.34375):15.7265625):16.1328125;

4.1.2. Requerimientos Iníciales

Además de las características presentadas en la versión secuencial del programa para obtener la filogenia de un grupo de secuencias, se hace necesario que:

- ❖ pyUPGMA, distribuya la carga de procesamiento adecuadamente entre los procesadores disponibles
- ❖ pyUPGMA, determine la filogenia para un grupo de secuencias en un menor tiempo al empleado por la version secuencial

4.2. DISEÑO

4.2.1. Partición

En pyUPGMA se usó tanto descomposición de dominio como funcional. En el primer caso dividimos los datos, secuencias, para el segundo dividimos las tareas, cálculos. La mejor descomposición será hallar distancias mínimas locales sobre la matriz de distancias, lo que nos permitirá obtener la menor redundancia de datos y la mayor concurrencia posible.

Cada tarea es una fila completa de la matriz y será responsable por el computo relacionado con ella, la cantidad de tareas creadas es igual al número de filas de la matriz de la iteración actual.

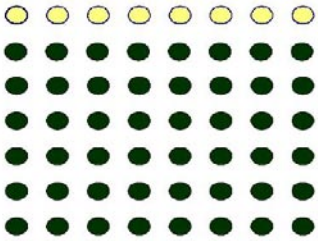


Fig. 4.1. Fila en color amarillo que representa una tarea completa.

En la medida en que la matriz va disminuyendo su tamaño por la iteración del algoritmo el numero de tareas ira disminuyendo igualmente. En total el numero de tareas será igual a la suma de n + (n-1) + (n-2) + (n-3) + ... + (n-i). Donde n es el total de filas, y la diferencia (n-i) es igual a dos.

4.2.2. Comunicación

El tipo de comunicación es completamente global (no existe ningún intercambio de información entre nodos); El maestro luego de cargar las secuencias las envía a los nodos para que efectúen diferentes operaciones sobre ellas, como calculo de distancias y determinación de distancias mínimas locales. La siguiente comunicación se efectúa cuando todos los nodos han realizado el cálculo de su subproblema, enviando el resultado al maestro para que lo organice de tal forma que genere la solución completa.

4.2.3. Agrupación

En la primera etapa se llevo a cabo la descomposicion, la cual permite determinar el numero de tareas a efectuar, se puede ver que el numero de tareas crece con el tamaño del problema, y que el numero de tareas sera de acuerdo al numero de filas de la matriz generada en cada iteracion; En esta tercera etapa del diseño de algoritmos paralelos, se considera util combinar las tareas identificadas en la etapa de particionamiento. pyUPGMA agrupa por filas logrando conservar la flexibilidad y obteniendo una mejor distribucion del trabajo.

4.2.4. Asignación

La asignación realizada en pyUPGMA se basa en un esquema de balanceo de carga Maestro/Esclavo. Asignando un número de tareas a cada nodo acorde a la agrupacion hecha, en otras palabras deberia ser de la siguiente forma: numero de filas dividido por el numero de nodos. Dado que la mayoria de las veces habra residuos en esta división la asignacion sera lo mas equitativa posible entre los nodos.

La asignacion usada en pyUPGMA define un valor inicial y uno final para cada nodo, asi se asignan valores enteros a los nodos sin dejar residuos; el valor inicial y final son calculados con las siguientes líneas de seudocódigo:

```
Valor_inicial=(parte_entera(longitu_de_la_matriz/numero_de_procesadores-
1) *(id_del_procesador-1))+1

Valor_final=parte_entera(longitude_de_la_matriz*((id_del_procesador)/(num
ero_de_procesadores-1)))
```

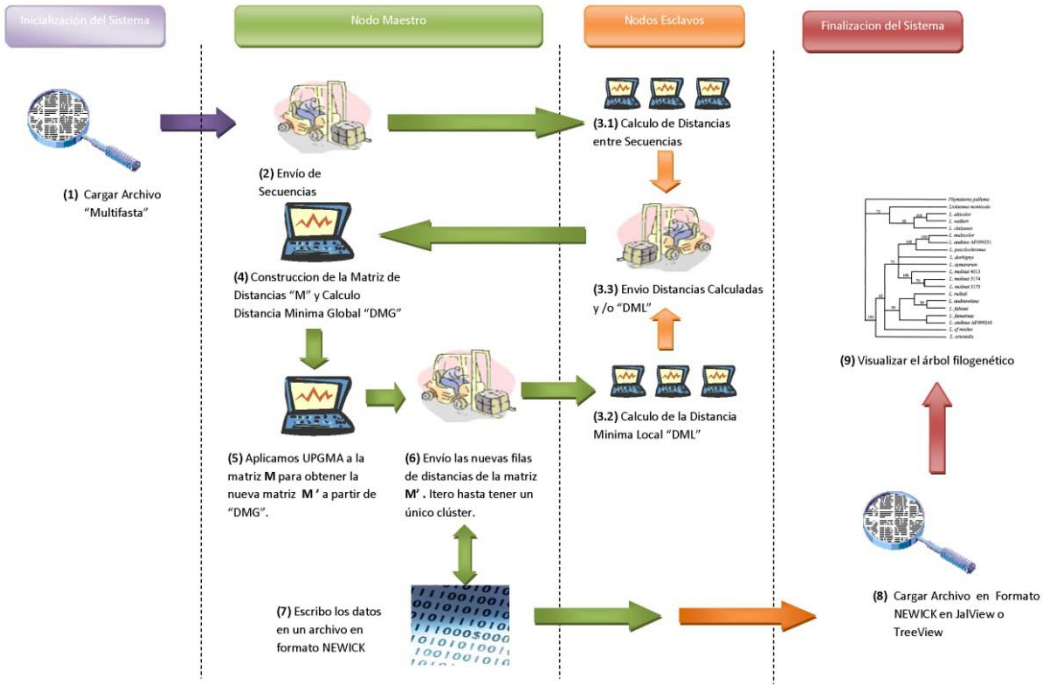


Fig. 4.2. Esquema de Funcionamiento de pyUPGMA

4.3. IMPLEMENTACION

Luego de aplicar la metodología de Ian Foster para el diseño de aplicaciones paralelas, se procede a codificar pyUPGMA y depurar las funciones para que trabaje en forma paralela.

4.3.1. Lenguaje de Programación

El lenguaje de programación empleado fue Python 2.4 para Linux, y la libreria pyMPI-2.4b4 para Python, esta ultima encargada de brindar el ambiente de desarrollo paralelo sobre Python.

Respecto a la selección de pyMPI como libreria de paso de mensajes se tuvo en cuenta soporte técnico, también que es la mejor y mas sencilla de usar entre las otras distribuciones (MyMPI, pyPAR,PP) de MPI para Python y ademas el tipo de maquina donde se ejecutaría la aplicación esta basada en el paradigma de memoria distribuida.

4.3.2. Formato de Datos de Entrada

Los datos de entrada conservan el formato de la versión secuencial, adicionalmente a estos se agrega un nuevo parámetro que nos indicara el número de procesadores que se van a utilizar para ejecutar la aplicación paralela.

- Versión Secuencial:

  ```
  [jairodsl@genoma ~]$python pyUPGMAs.py <archivo_fasta> <tipo_seq>
  ```

- Versión Paralela:

```
[jairodsl@genoma ~]$./pyMPI <archivo_fasta> <tipo_seq> <num_procs>
```

En este prototipo final paralelo, pyUPGMA se maneja desde la consola de comandos. Para futuras versiones se recomienda una interfaz grafica sobre ambiente web, como actualmente están implementando todas las aplicaciones bioinformáticas.

4.3.3. Implementación del Algoritmo

pyUGMA está dividido en un programa principal y librerías acorde al tipo de secuencia que deseamos procesar. Alguna de ellas puede requerir el uso de otra librería para algún cálculo específico.

Tabla. 4.1. Librerías que conforman pyUPGMA

Programa / Librería	Descripción
pyUPGMA	Programa principal
upgmaNuc	Libreria que almacena las funciones para el trabajo con nucleotidos
upgmaAmi	Libreria que almacena las funciones para el trabajo con aminoacidos
Blosum62	Libreria que almacena las funciones para la construcción de una matriz de distancias usando Blosum62 para aminoacidos.

A continuación se describen los cambios en el programa principal y las librerias que permitieron la paralelización de UPGMA.

- ***pyUPGMA***

Selección de un nodo maestro (mpi.rank==0) encargado de recibir los datos de entrada del usuario y los nodos esclavos (mpi.rank!=0) encargados de realizar las tareas asignadas. Al final cada uno de los nodos esclavos regresa al maestro el resultado de la tarea asignada.

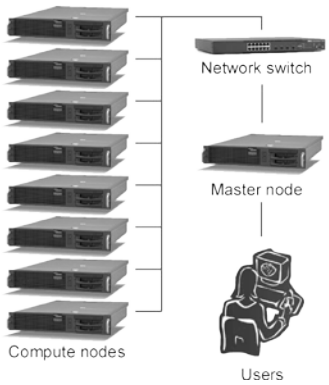


Fig. 4.3. Esquema Maestro/Esclavo

Dos tipos de modificaciones al programa principal fueron encontradas: modificaciones a nivel global, donde la lectura de los parámetros iníciales es realizada por el nodo maestro, y modificaciones a nivel administrativo, tales modificaciones se traducen en la creacion de nuevas variables que permitirán el debido intercambio de información entre los nodos esclavos y el maestro.

La tabla 4.2 muestra en seudocódigo las modificaciones del programa principal para poder paralelizar pyUPGMA, en la tabla 4.3 se muestran las nuevas variables necesarias para la paralelización.

Tabla. 4.2. Modificaciones del programa principal en seudocodigo

```
Sí (mi_id es igual a 0) entonces:
        Lea_el_archivo_multifasta
        nombres_de_las_secuencias=nombres
        lista_de_las_secuencias=secuencias
        broadcast(secuencias)
        broadcast(nombres)
Si (mi_id no es igual a 0) entonces:
        receive(secuencias)
        receive(nombres)
        Sí (tipo_secuencia es Nucleotido) hacer:
        import upgmaNuc
        Sí  (tipo_secuencia es Aminoacido) hacer:
        import upgmaAmi
```

```
Sí (mi_id no es igual a 0) entonces:
        Función vivf:
                Hallar_vi_para_cada_nodo
                Hallar_vf_para_cada_nodo
        Regresar valores_iniciales_y_finales
        Send(filas_de_distancias,maestro)
Sí (mi_id es igual a 0) entonces:
        matriz=[]
        receive(filas_de_distancias)
        matriz[fila_de_distancia]=fila_de_distancia[valor]
        print matriz
```

```
Sí (mi_id es igual a 0) entonces:
        Mientras nombres>1 hacer:
                Send(matriz)
                distancias_minimas=[]
              Receive(distancias_minimas)
                distancia_global_min=minimo(distancias_minimas)
                iteracion=UPGMA(matriz,nombres,distancia_global_min)
                matriz=iteración[nueva_matriz_resultante
               nombres=iteración[nuevos_nombres]
        print Filogenia_Obtenida
Sí (mi_id no es igual a 0):
        Mientras nombres>1 hacer:
                receive(matriz)
                rango=vivf(matriz) #Valores inciales y finales
                Si (rango es  correcto) hacer:
                menor(matriz,rango)
                send(menor)
                Si (rango no es correcto) hacer:
                send(dummy)
```

Tabla. 4.3. Nuevas Variables del Programa Principal

Variables	Función
mpi.rank	Almacena el identificador de cada procesador
mpi.size	Almacena el numero total de procesadores disponibles
e_sec	Almacena las secuencias enviadas desde el maestro contenidas en m_sec
e_nom	Almacena los nombres de las secuencias enviadas desde el maestro contenidas en m_nom
e_mat	Almacena las filas de la matriz de distancias para cada nodo acorde con las secuencias e_sec
Vi	Almacena el valor inicial del numero de filas a procesar
Vf	Almacena el valor final del numero de filas a procesar
m_mat	Almacena la matriz de distancias a partir de las filas e_mat de cada nodo esclavo
Dml	Almacena las distancias minimas locales obtenidas por cada uno de los nodos
Dmg	Almacena la distancia minima global entre las distancias minimas locales
m_matrix	Almacena la matriz que va cambiando con la iteracion de UPGMA
m_nom	Almacena la filogenia a medidad que va iterando el algoritmo UPGMA sobre la matriz m_matrix
e_menor	Almacena la menor distancia y su posicion en la matriz, para cada nodo esclavo.

- ***upgmaNuc y upgmaAmi***

Estas librerías tienen un pequeño cambio en las mismas funciones para ambos casos, las funciones que cambian son la función menor y upgma, la primera determina la menor distancia y su posición en una matriz. Tal función necesita conocer un rango inicial y final calculado de acuerdo al id de procesador. En la función upgma ahora agregamos la distancia mínima global de cada iteración.

En las tablas 4.4. a 4.6. las funciones “menor” y “upgma” antes y después, en seudocódigo.

Tabla 4.4. Función Menor secuencial

Función	Antes
menor	def menor(matriz): Dummy←1e300 Posición← almacena la ubicación en la matriz de la menor distancia Para i y j en la matriz: Sí (índices de fila y columna son diferente) hacer: Sí (elemento_ij mayor o igual que 0) y (elemento_ij es mayor que dummy) hacer: Dummy←elemento_ij posición ←índices del elemento_i regresar dummy,posición

Tabla 4.5. Función Menor Modificada para Trabajar en Paralelo

Función	Después
Menor	def menor(matriz,rango): dummy←1e300 Para i y j en la matriz: Para rango en la matriz: Sí (índices fila y columna son diferentes) hacer: Sí (elemento[i][j] ➔ rango >= 0) and (elemento[i][j] ➔ rango >dummy) hacer: dummy←elemento[i][j] posición ←coordenadas_elemento[i][j] regresar dummy,posición

Tabla 4.6. Función Upgma secuencial

Función	Antes
Upgma	def upgma(matriz,nombres): emenor=menor(matriz) índices_a_borrar=emenor[i], emenor[j] índices=índices – índices_a_borrar nombres.pop(índices_a_borrar) nombres[índices]=nombres matriz2=matriz upgma2=[] Para índices en matriz2: matriz2.pop(índices_a_borrar) Para índices en matriz2: upgma2=(matriz2[índice]+matriz2[índice])/2.0 matriz2.pop(índices_a_borrar) matriz2.insert(upgma2) return maux,nombres

Tabla. 4.7 Función Upgma Modificada para Trabajar en Paralelo

Función	Despúes
upgma	def upgma(matriz,nombres,distancia_minima_global): emenor=distancia_minima_global índices_a_borrar=emenor[i], emenor[j] índices=índices – índices_a_borrar nombres.pop(índices_a_borrar) nombres[índices]=nombres matriz2=matriz upgma2=[] Para índices en matriz2: matriz2.pop(índices_a_borrar) Para índices en matriz2: upgma2=(matriz2[índice]+matriz2[índice])/2.0 matriz2.pop(índices_a_borrar) matriz2.insert(upgma2) return maux,nombres

- ***Blosum62***

A esta librería no se le realizaron cambios, dado que no estaba dentro del alcance del proyecto. De manera independiente de este proyecto deberia paralelizarse esta librería.

5. PRUEBAS

El objetivo en esta fase de pruebas es valorar del programa desarrollado en este proyecto la eficiencia a través de una librería de Python llamada Profile por medio de un análisis deterministico del tiempo empleado y el número de veces que las funciones son ejecutadas dentro del programa. Además se determina el rendimiento de los tiempos de respuesta del algoritmo UPGMA cuando se ejecuta en secuencial versus los tiempos de respuesta del mismo algoritmo en paralelo, evaluando la Ley de Amdahl.

5.1. PYTHON PROFILER

Un Profiler (http://docs.python.org/lib/profile.html) es una aplicación que describe en el tiempo de ejecución de un programa, proveyendo una variedad de estadísticas. Este profiler provee una serie de herramientas de generación de reportes que permite a los usuarios examinar rápidamente los resultados de una operación de profiling[10].

- Profile en Acción

Existen varias formas de ejecutar la librería Profile, para los casos estudiados se hizo invocandola directamente desde el interprete de comandos y asignando un nombre al archivo binario que contendrá las estadísticas:

```
python -m profile -o <archivo_resultados> <programa_python_a_analizar>
```

En tal caso para ver que sucede dentro de pyUPGMA secuencial cuando esta determinando la filogenia para el HIV1 la línea de ejecución seria como sigue:

```
python -m profile -o perfil_hiv1.dat pyUPGMA.py
```

El archivo perfil_hiv1.dat es un archivo binario que solo se puede analizar desde Python de la mano de la librería PStats, complemento de la librería Profile. La sintaxis para visualizar el archivo obtenido es:

```
import pstats
p=pstats.Stats('perfil_hiv1.dat')
p.print_stats()
```

Para un mejor detalle el resultado se ordena por el numero de llamadas hechas a cada componente del programa, como se puede ver en la grafica 5.1, la sintaxis para ordenar y visualizar los datos es como sigue:

[10] http://docs.python.org/lib/node795.html

```
p.sort_stats('calls').print_stats()
```

No olvidar que la variable p en este caso es una instancia de la clase pstats.

```
>>> p.sort_stats('calls').print_stats()
           11:55:14 2007    perfil_hiv1.dat

         1604 function calls in 0.088 CPU seconds

   Ordered by: call count

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
      451    0.003    0.000    0.003    0.000 :0(len)
      334    0.002    0.000    0.002    0.000 :0(append)
      246    0.003    0.000    0.003    0.000 :0(range)
      228    0.002    0.000    0.002    0.000 :0(pop)
      169    0.028    0.000    0.032    0.000 upgmaNuc.py:62(compfasta)
      114    0.001    0.000    0.001    0.000 :0(insert)
       12    0.001    0.000    0.001    0.000 upgmaNuc.py:87(menor)
       12    0.000    0.000    0.000    0.000 :0(sort)
       12    0.014    0.001    0.022    0.002 upgmaNuc.py:106(upgma)
        4    0.000    0.000    0.000    0.000 :0(replace)
        2    0.001    0.000    0.001    0.000 :0(open)
        2    0.000    0.000    0.000    0.000 :0(time)
        2    0.000    0.000    0.000    0.000 :0(close)
        1    0.000    0.000    0.000    0.000 upgmaNuc.py:22(<module>)
        1    0.004    0.004    0.087    0.087 :0(execfile)
        1    0.000    0.000    0.088    0.088 profile:0(execfile('pyUPGMAs.py'))
        1    0.000    0.000    0.001    0.001 upgmaNuc.py:167(salida)
        1    0.003    0.003    0.036    0.036 upgmaNuc.py:74(matriz)
        1    0.001    0.001    0.001    0.001 :0(setprofile)
        1    0.000    0.000    0.000    0.000 :0(split)
        1    0.000    0.000    0.023    0.023 upgmaNuc.py:143(repite)
        1    0.000    0.000    0.000    0.000 :0(read)
        1    0.000    0.000    0.000    0.000 upgmaNuc.py:40(creanom)
        1    0.023    0.023    0.083    0.083 pyUPGMAs.py:6(<module>)
        1    0.000    0.000    0.000    0.000 upgmaNuc.py:155(fNewick)
        1    0.000    0.000    0.000    0.000 :0(write)
        1    0.000    0.000    0.000    0.000 upgmaNuc.py:22(leamultif)
        1    0.000    0.000    0.000    0.000 upgmaNuc.py:51(creasec)
        1    0.000    0.000    0.087    0.087 <string>:1(<module>)
        0    0.000             0.000          profile:0(profiler)


<pstats.Stats instance at 0x00F5D238>
>>>
```

Fig. 5.1. Estadísticas de la Herramienta Profiler al calcular la filogenia para HIV1.

- En la figura 5.1 se observa que básicamente son cinco funciones definidas por python las más ejecutadas, es decir: len, append, range, pop, insert, sumando un total de 1373 llamadas, pero en un tiempo de cálculo muy corto, apenas de solo

0.011 segundos. Este tiempo representa el 12.5% aproximadamente del tiempo total empleado

- Las funciones implementadas en este programa que más llamadas genera son tres: compfasta, menor y upgma, para un total de 193 llamadas, en cambio si consumen más tiempo que las funciones propias de python, sumando 0.055 segundos. Este tiempo representa el 62.5% aproximadamente del tiempo total empleado.
- La función matriz, por si sola es llamada una vez, pero tiene un alto tiempo de ejecución, de 0.024 segundos. El tiempo empleado por la función matriz en el caso del HIV-1 representa el 25% del tiempo total de ejecución.

Como se menciono en el planteamiento del problema en el capítulo 1 la filogenia es computacionalmente demandante, a medida que aumenta el número de especies, lo cual veremos en las estadísticas obtenidas para el HPV, a continuación.

```
>>> p.sort_stats('calls').print_stats()
              10:57 2007    perfil_hpv.dat

         80540 function calls in 2.347 CPU seconds

   Ordered by: call count

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
    22991    0.135    0.000    0.135    0.000 :0(len)
    17216    0.092    0.000    0.092    0.000 :0(append)
    11654    0.087    0.000    0.087    0.000 :0(range)
    11544    0.101    0.000    0.101    0.000 :0(pop)
    11025    0.313    0.000    0.451    0.000 upgmaNuc.py:62(compfasta)
     5772    0.047    0.000    0.047    0.000 :0(insert)
      104    0.264    0.003    0.265    0.003 upgmaNuc.py:87(menor)
      104    0.001    0.000    0.001    0.000 :0(sort)
      104    0.376    0.004    0.908    0.009 upgmaNuc.py:106(upgma)
        4    0.000    0.000    0.000    0.000 :0(replace)
        2    0.013    0.007    0.013    0.007 :0(open)
        2    0.000    0.000    0.000    0.000 :0(time)
        2    0.000    0.000    0.000    0.000 :0(close)
        1    0.000    0.000    0.000    0.000 upgmaNuc.py:22(<module>)
        1    0.003    0.003    2.346    2.346 :0(execfile)
        1    0.000    0.000    2.347    2.347 profile:0(execfile('pyUPGMAs.py'))
        1    0.000    0.000    0.013    0.013 upgmaNuc.py:167(salida)
        1    0.141    0.141    0.646    0.646 upgmaNuc.py:74(matriz)
        1    0.001    0.001    0.001    0.001 :0(setprofile)
        1    0.000    0.000    0.000    0.000 :0(split)
        1    0.002    0.002    0.911    0.911 upgmaNuc.py:143(repite)
        1    0.021    0.021    0.021    0.021 :0(read)
        1    0.001    0.001    0.001    0.001 upgmaNuc.py:40(creanom)
        1    0.748    0.748    2.343    2.343 pyUPGMAs.py:6(<module>)
        1    0.000    0.000    0.000    0.000 upgmaNuc.py:155(fNewick)
        1    0.000    0.000    0.000    0.000 :0(write)
        1    0.000    0.000    0.022    0.022 upgmaNuc.py:22(leamultif)
        1    0.001    0.001    0.001    0.001 upgmaNuc.py:51(creasec)
        1    0.000    0.000    2.346    2.346 <string>:1(<module>)
        0    0.000             0.000          profile:0(profiler)


<pstats.Stats instance at 0x00E80A80>
>>>
```

Fig. 5.2. Estadísticas de la Herramienta Profiler al calcular la filogenia para HPV.

- Al revisar los resultados en la figura 5.2 se observa que las funciones propias de python más ejecutadas, son: len, append,range, pop, e insert, suman ahora 69177 llamadas en total, dado que el numero de secuencias ahora analizadas son 105 para HPV contra 13 del HIV1. Y el tiempo que emplean en este caso es de 0.462 segundos, aproximadamente el tiempo se incremento en un factor de 40. Este tiempo representa aproximadamente el 20% del tiempo total de ejecución.
- Las funciones compfasta, menor y upgma, totalizaron 11263 llamadas. Pasando de un tiempo de 0.055 segundos para HIV1 a 1.624 segundos para el HPV, aumentando aproximadamente en un factor de 30. El tiempo empleado por las funciones compfasta, menor y upgma en el caso del HPV representa el 69% con respecto al tiempo total de ejecución
- La función matriz para el HPV empleo 0.381 segundos, aproximadamente se aumento en un factor de 15 comparado con los 0.024 segundos que empleo para HIV1. El tiempo empleado de la función matriz en el caso del HPV representa aproximadamente el 11% del tiempo total de ejecución.

5.2. LEY DE AMDAHL

Dos de las medidas más importantes para apreciar la implementación de un algoritmo paralelo en multicomputadores y multiprocesadores son aceleración y eficiencia. La aceleración S de un algoritmo paralelo ejecutado usando *p* procesadores es la razón entre el tiempo que tarda el mejor algoritmo secuencial en ser ejecutado usando un procesador en un computador y el tiempo que tarda el correspondiente algoritmo paralelo en ser ejecutado en el mismo computador usando *p* procesadores.

$$S=\frac{Tiempo\ de\ Ejecución\ secuencial}{Tiempo\ de\ Ejecución\ paralela}$$

La eficiencia de un algoritmo paralelo ejecutado en *p* procesadores es la aceleración dividida por *p*. La ley de *Amdahl* permite expresar la aceleración máxima obtenible como una función de la fracción del código del algoritmo que sea paralelizable. Sea *T* el tiempo que tarda el mejor algoritmo secuencial en correr en un computador. Si *f* es la fracción de operaciones del algoritmo que hay que ejecutar secuencialmente ($0 < f < 1$), entonces la aceleración máxima obtenible ejecutando el algoritmo en *p* procesadores es:

$$S=\frac{T}{T(f+\frac{1-f}{p})}=\frac{1}{f+(1-f)/p}$$

En el limite, cuando $f \rightarrow 0$, $S = p$. Excepto casos muy particulares, este límite nunca es obtenible debido a las siguientes razones:

- inevitablemente parte del algoritmo es secuencial
- hay conflictos de datos y memoria
- la comunicación y sincronización de procesos consume tiempo
- es muy difícil lograr un balanceo de carga perfecto entre los procesadores.

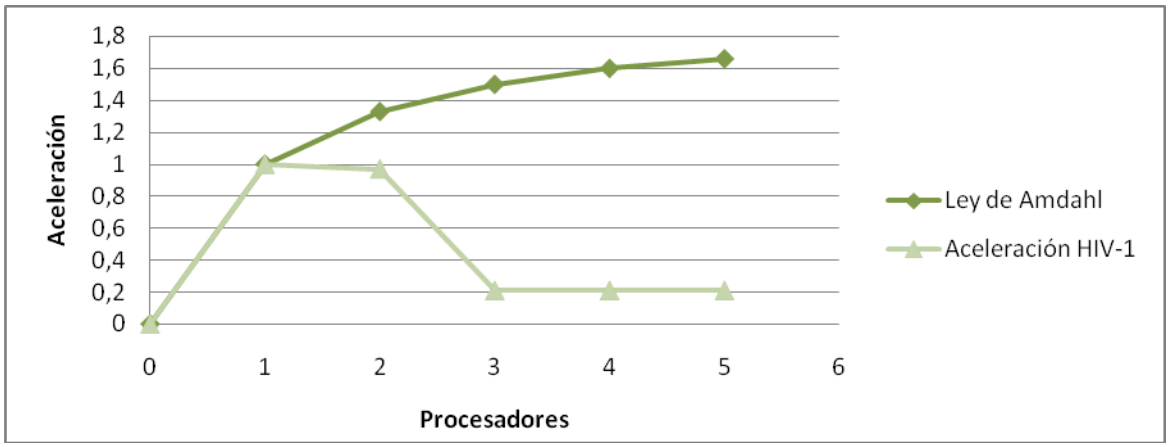


Fig. 5.3. Aceleración obtenida para el cálculo de la Filogenia para el HIV-1.

En la figura 5.3 se observa la aceleración obtenida para el cálculo de la filogenia del HIV-1. En este caso no se obtiene una buena medida de la aceleración, con dos procesadores se logra estar cerca del mismo tiempo que para un procesador. El modelo como se puede ver no es de gran tamaño y usar más procesadores para tan pocos datos no alcanza la aceleración ideal y en cambio produce una caída de la aceleración notoria, pues cada vez va aumentar el costo de comunicación y el número de operaciones realizadas sobre los datos que cada procesador recibe son pocas, por lo que se castiga el rendimiento de la aplicación paralela.

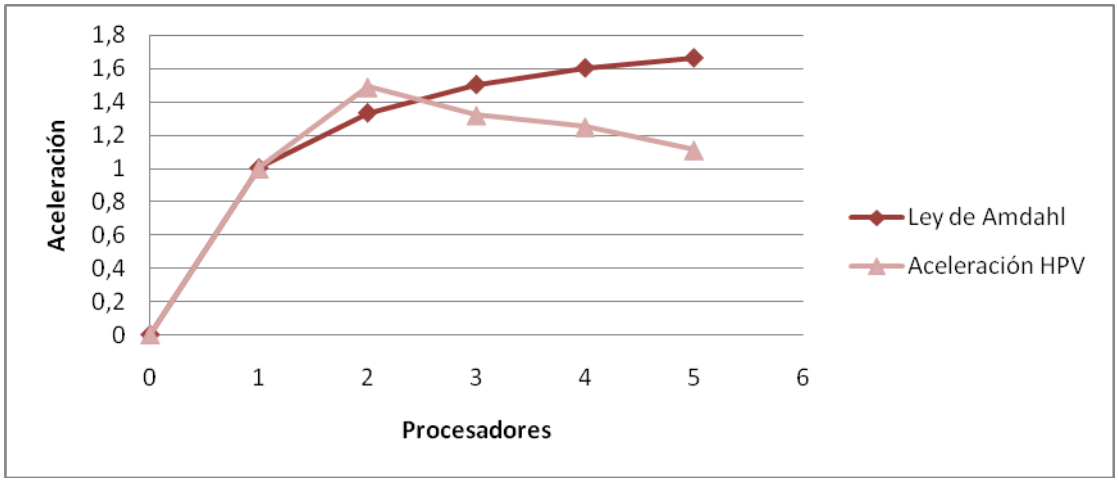


Fig. 5.4. Aceleración obtenida para el cálculo de la Filogenia para el HPV.

Como observamos en la grafica 5.4., el modelo del Papiloma Virus – HPV es mucho más grande que el del HIV-1, y se destaca una aceleración sustancial en el calculo de la filogenia, pero a medida que se incrementa el número de procesadores la aceleración disminuye acercándose a la aceleración para un procesador. Estas aceleraciones se determinaron para ambos modelos tomando como base que el 50% del código se paralelizo.

Si normalizamos la aceleración dividiéndola por el número de procesadores obtenemos la eficiencia $\eta = S/p$. Luego a medida que se incrementa el número de procesadores p, la eficiencia η decrementa. Esto refleja el hecho de que a medida que el código es dividido en trozos más y más pequeños, eventualmente algunos procesadores deben permanecer

ociosos esperando que otros terminen ciertas tareas. En la práctica esto no es así, ya que solo a medida que los problemas se hacen más grandes es que se asignan a más procesadores.

Si asumimos que el código es 100% paralelizable y por lo tanto la aceleración es $S = p$; despreciando el tiempo de comunicación entre procesadores, considerando el *tiempo de comunicación* o *latencia Tc* , la aceleración decrece aproximadamente a

$$S=\frac{T}{T/p+Tc}<p$$

y para que la aceleración no sea afectada por la latencia de comunicación necesitamos que:

$$\frac{T}{p} \gg Tc \rightarrow p<<\frac{T}{Tc}$$

Esto significa que a medida que se divide el problema en partes más y más pequeñas para poder correr el problema en más procesadores, llega un momento en que el costo de comunicación *Tc* se hace muy significativo y desacelera el cómputo.

5.3. PLATAFORMA DE EJECUCIÓN

Las pruebas se llevaron a cabo en varios tipos de clúster. Se uso un clúster virtual de 4 nodos usando la herramienta de virtualizacion VMWARE 5.3 sobre Windows XP con S.O Fedora 5, con 256 Mb de Ram y 5 Gigas de Espacio para cada nodo; se uso con el objetivo de comprobar la funcionalidad los prototipos antes de ejecutarlos en el cluster físico Genoma y no desperdiciar recursos ni tiempo en la carga de archivos al cluster fisico.

La plataforma principal fue el Cluster Genoma con 4 nodos de 1 giga de Ram, procesadores de 2.6ghz Pentium 4, disco duro de 80gb y S.O. Linux Rocks 4.2.1 con red Ethernet; esta fue usada por medio de acceso seguro con protocolo SSH con un id y password asignado por CeCalcULA.

6. RESULTADOS

En la fase iterativa de este proyecto se implementaron prototipos secuencial y paralelo del algoritmo Unweighted Pair Group Method with Arithmetic mean – UPGMA, sobre plataforma Windows/Linux, pero solo la versión paralela es posible ejecutarla en Linux, según el listado de http://www.top500.org las aplicaciones paralelas tienen mejor rendimiento sobre Linux. Sin embargo la librería de paso de mensajes pyMPI para Python está dedicado en su totalidad a ambientes Linux.

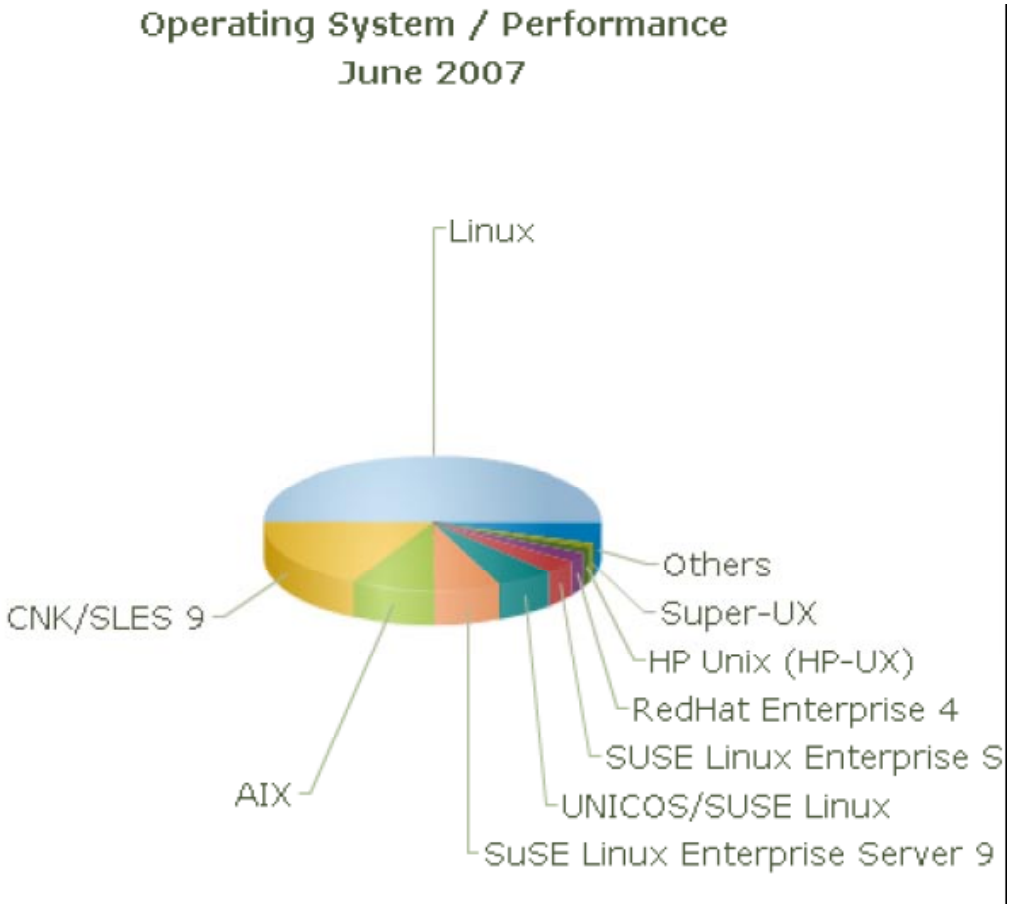


Fig. 6.1. Grafica del Rendimiento de Aplicaciones Paralelas según el sistema operativo. Fuente: http://www.top500.org

Para determinar la funcionalidad del prototipo secuencial se uso una prueba tomada de la web http://www.diku.dk/~pawel/comp-bio/ev_trees/intro/upgma_ex.html [11], de la cual se

[11] Winter, Pawel. Associate Professor, Dept. of Computer Science University of Copenhage, Denmark

usó la matriz de distancias para reconstruir la filogenia entre las siguientes especies: Perro, Oso, Mapache, Comadreja, Foca, León marino, Gato y Mono. Para el prototipo paralelo se usaron dos pruebas que dejase en claro el rendimiento de pyUPGMA secuencial y paralelo, las cuales corresponden a pruebas con los virus del inmunodeficiencia humana Tipo 1 y al Papilloma Humano.

¿Por qué analizar filogenéticamente un virus?

Los análisis filogenéticos se utilizan esencialmente en todas las ramas de la biología; el rango de aplicaciones va desde el estudio del origen de poblaciones humanas hasta realizar investigaciones de importantes preguntas sobre epidemiologia. Los análisis filogenéticos se utilizan cada vez más en intentos por clarificar patrones de transmisión de virus, como por ejemplo el virus de inmunodeficiencia humana del tipo 1. Pero hay una continua discusión sobre su validez porque la evolución convergente y menores variantes del HIV-1 pueden oscurecer los patrones epidemiológicos[12] .

6.1. PRUEBA BASICA

Como se describió al comienzo la prueba básica está basada en el ejemplo tomado de la página web del profesor Pawel Winter disponible en http://www.diku.dk/~pawel.

El modelo a seguir tiene la siguiente matriz de distancias:

Tabla 6.1. Matriz de Distancias

	Perro	Oso	Mapache	Comadreja	Foca	León Marino	Gato	Mono
Perro	0	32	48	51	50	48	98	148
Oso		0	26	34	29	33	84	136
Mapache			0	42	44	44	92	152
Comadreja				0	44	38	86	142
Foca					0	24	89	142
León Marino						0	90	142
Gato							0	148
Mono								0

Si aplicamos UPGMA manualmente obtendremos la primera ramificación del árbol filogenético:

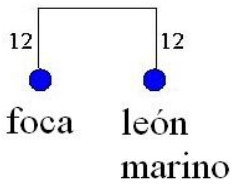


Fig. 6.2. Primera ramificación del Arbol para la Prueba Básica

[12] "Accurate reconstruction of a knownHIV-1 transmission history by phylogenetic tree analysis" by Thomas Leitner, David Escanilla, Christer Franzén, Mathias Uhlén, and Jan Albert.

Dentro del prototipo secuencial de pyUPGMA la matriz de distancias para este ejemplo tiene la siguiente estructura:

```
matriz=[[0,32,48,51,50,48,98,148],[32,0,26,34,29,33,84,136],[48,26,0,42,4
4,44,92,152],[51,34,42,0,44,38,86,142],[50,29,44,44,0,24,89,142],[48,33,4
4,38,24,0,90,142],[98,84,92,86,89,90,0,148],[148,136,152,142,142,142,148,
0]]
```

Y la primera ramificación se vera de la siguiente forma en pyUPGMA:

```
['perro', 'oso', 'mapache', 'comadreja', 'gato', 'mono', '(foca,león
marino):12.0']
```

Luego de seis iteraciones más tenemos la matriz definitiva:

Tabla 6.2. Matriz final

	((((Foca,Leon Marino),(Oso,Mapache),Comadreja), Perro).Gato)	Mono
((((Foca,Leon Marino),(Oso,Mapache),Comadreja), Perro).Gato)	0	144.2
Mono		0

El árbol filogenético luego de todas las iteraciones es como sigue:

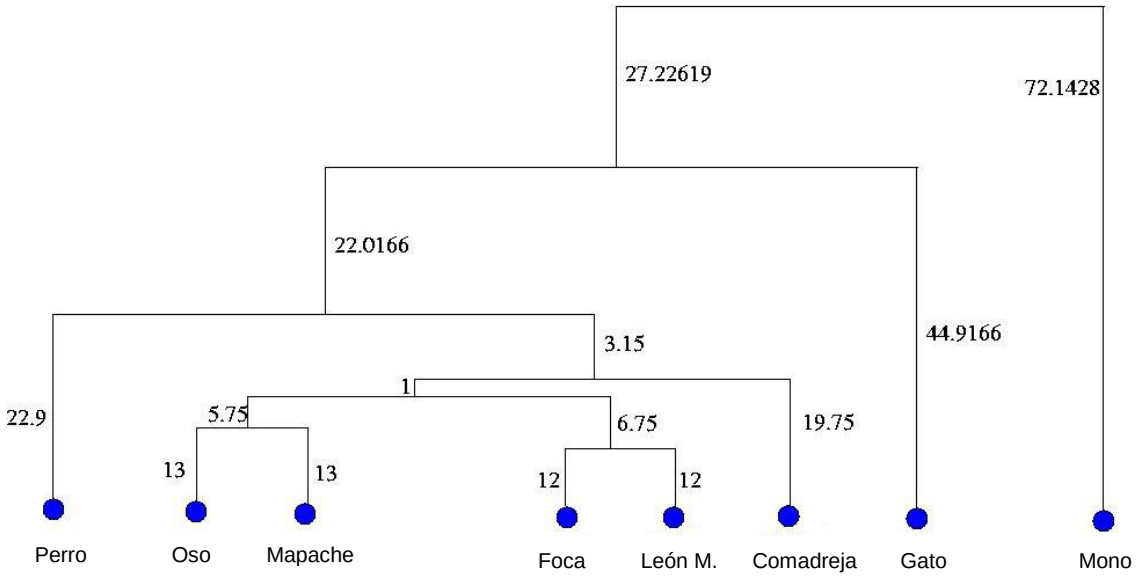


Fig. 6.3. Arbol Filogenético completo para la Prueba Básica

La matriz final en pyUPGMA será la siguiente:

```
[0, 146.625]
[146.625, 0.0]
```

El árbol filogenético final obtenido con el programa pyUPGMA en el formato Newick es:

```
['(mono,(gato,(perro,(comadreja,((foca,leonmarino):12.0,(oso,mapache):13.
0):18.75):19.75):23.875):46.34375):73.3125']
```

Por tanto:

- El prototipo secuencial de pyUPGMA cumple efectivamente con el cálculo de la filogenia, como si de forma manual se hubiese hecho.
- El modelo empleado en esta prueba básica no es un modelo significativo, solo es un buen modelo para comprobar la funcionalidad de pyUPGMA secuencial y que el algoritmo está bien implementado.
- Durante la prueba básica no se hizo análisis de rendimiento, por las condiciones del modelo y el prototipo empleado.
- La diferencia en los valores de las distancias obtenidas se debe al número de cifras decimales tenidas en cuenta por el interprete de Python y al número de cifras decimales que se tengan en cuenta al hacer el cálculo de forma manual.

6.2. PRUEBA DEL HIV-1

Para esta prueba de pyUPGMA tanto secuencial como en la versión paralela se usaron 13 secuencias de HIV-1 (es decir desde U68496 a la U68508 que se encuentran disponibles en la base de datos NCBI). Dichas secuencias se almacenaron en formato FASTA y el nombre del archivo fue llamado “HIV1.fasta” (ver anexo). Además el modelo del HIV nos permitió crear un escenario de trabajo más amplio para medir rendimiento.

6.2.1. HIV-1 calculado con la versión de pyUPGMA en secuencial

Basandose en la librería time (disponible en la página web http://docs.python.org/lib/module-time.html) de python, se obtuvo una primera medida de rendimiento para la aplicación.

- *Librería Time de Python:*

  De la librería Time de Python se uso la función time() en pyUPGMA secuencial, luego de solicitar al usuario el nombre de archivo y el tipo de secuencia, es decir no se contabiliza el tiempo de ejecución hasta tanto no se carga el archivo.

  El tiempo de respuesta obtenido en segundos luego de la última iteración del algoritmo es de **0.764999 segundos** para el primer prototipo secuencial. Este prototipo corre sobre ambiente Windows.

Para el prototipo final de pyUPGMA secuencial se alcanza un mejor tiempo de respuesta **0.025394 segundos** para determinar la filogenia del HIV-1. Y este prototipo mejora notablemente al ejecutarse sobre ambiente Linux.

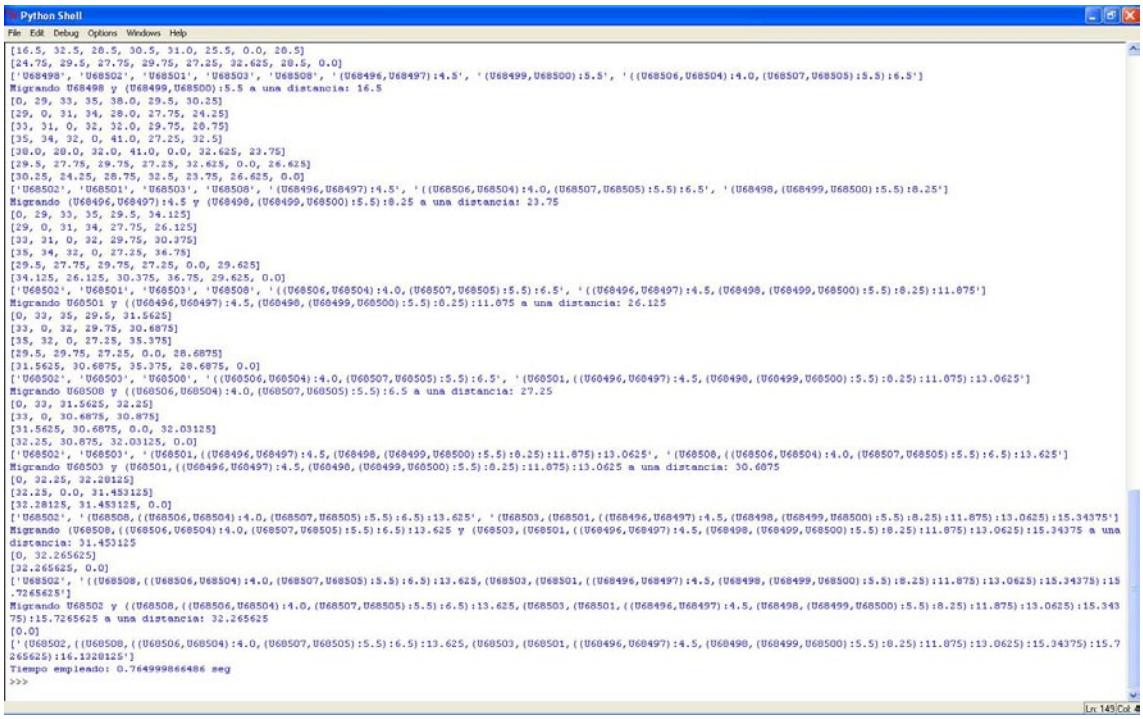


Fig. 6.4. pyUPGMA en ejecución, empleado para calcular la filogenia del HIV-1 en el primer prototipo secuencial sobre WindowsXP.

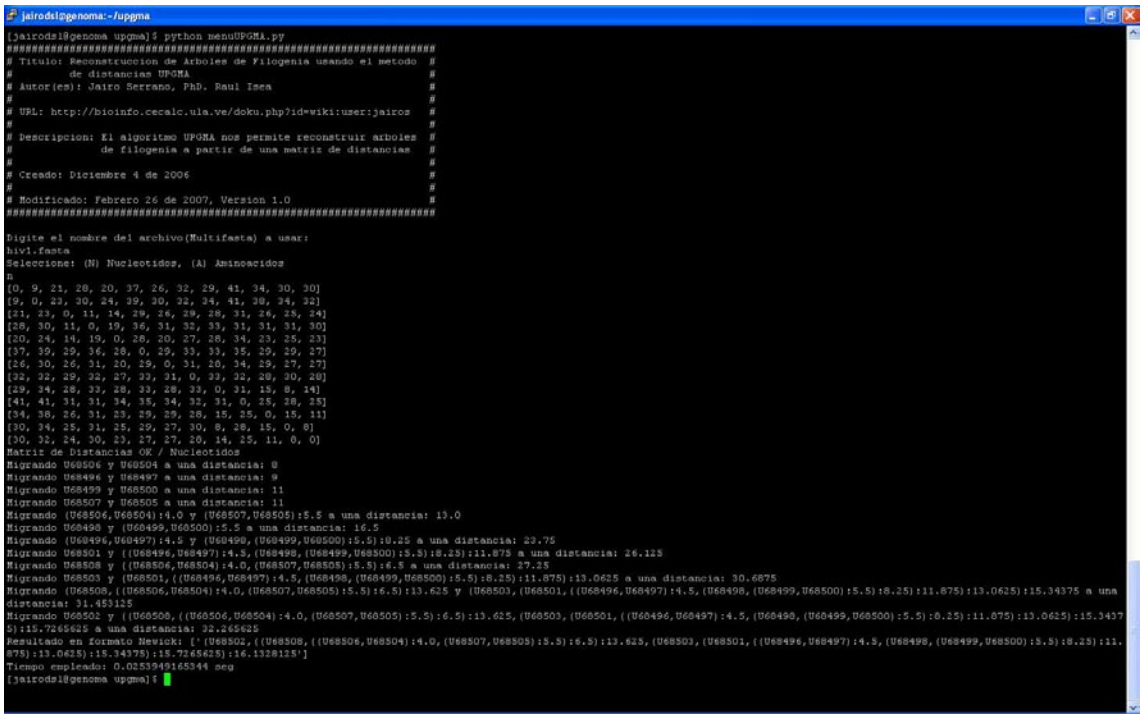


Fig. 6.5. pyUPGMA en ejecución, empleado para calcular la filogenia del HIV-1 en el primer prototipo secuencial sobre Linux.

Finalmente la visualización del árbol filogenético se realiza con el programa TreeView Treeview[13] con las secuencias HIV1.fasta. El archivo obtenido con la filogenia fue el mismo en todos los prototipos.

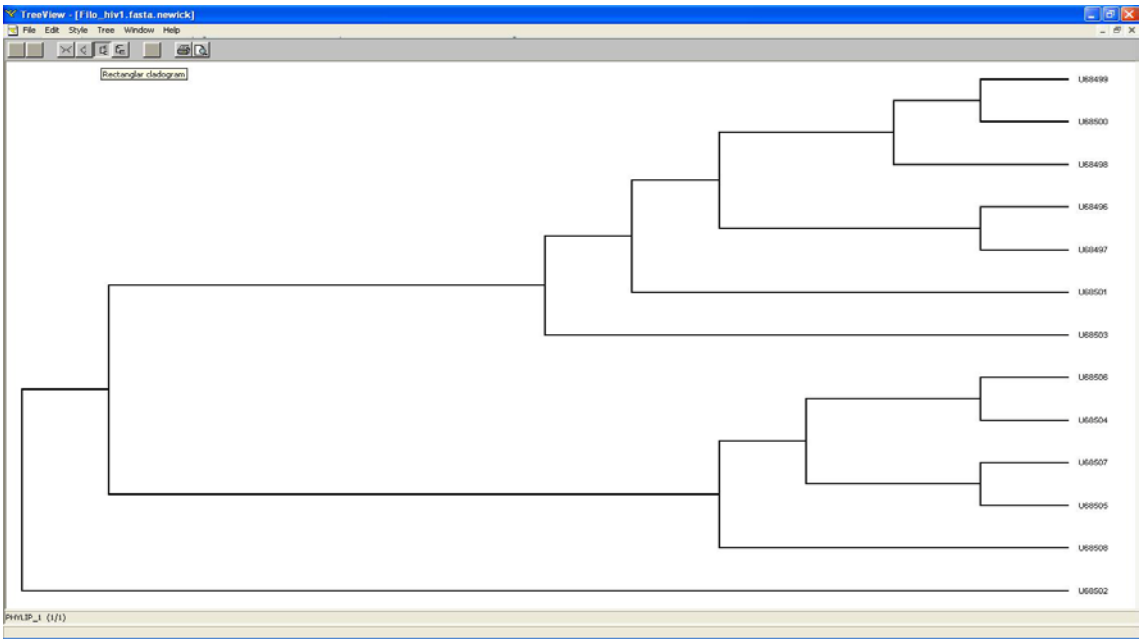


Fig. 6.6. Arbol Filogenético del HIV-1 visualizado con el programa Treeview.

6.2.2. HIV-1 calculado con la versión de pyUPGMA en paralelo

Las pruebas para pyUPGMA paralelo requirieron que el número de procesadores empleados sea mayor que dos, puesto que se diseño basado en el modelo Maestro/Esclavo. Aunque el prototipo paralelo inicial permitía usar dos procesadores no era justificable.

© *Función mpi.time() de pyMPI:*

En este caso para dos procesadores se obtuvo un tiempo de respuesta de **0.026167 segundos** contra 0.025394 segundos de la versión secuencial, este tiempo es aproximadamente 9% mayor que el secuencial. Este es un tiempo de respuesta muy parecido que no nos permite determinar si se justificó o no paralelizar UPGMA.

Si el número de procesadores crece a tres el tiempo de respuesta se incrementa, debido al gasto de comunicación entre los nodos esclavos y el nodo maestro. Para cuatro procesadores por ejemplo se obtuvo un tiempo de **0.116772 segundos.** Tiempo que aumento en un 22% aproximadamente respecto al secuencial.

[13] TreeView is a simple program for displaying phylogenies on Apple Macintosh and Windows PCs. http://taxonomy.zoology.gla.ac.uk/rod/treeview.html

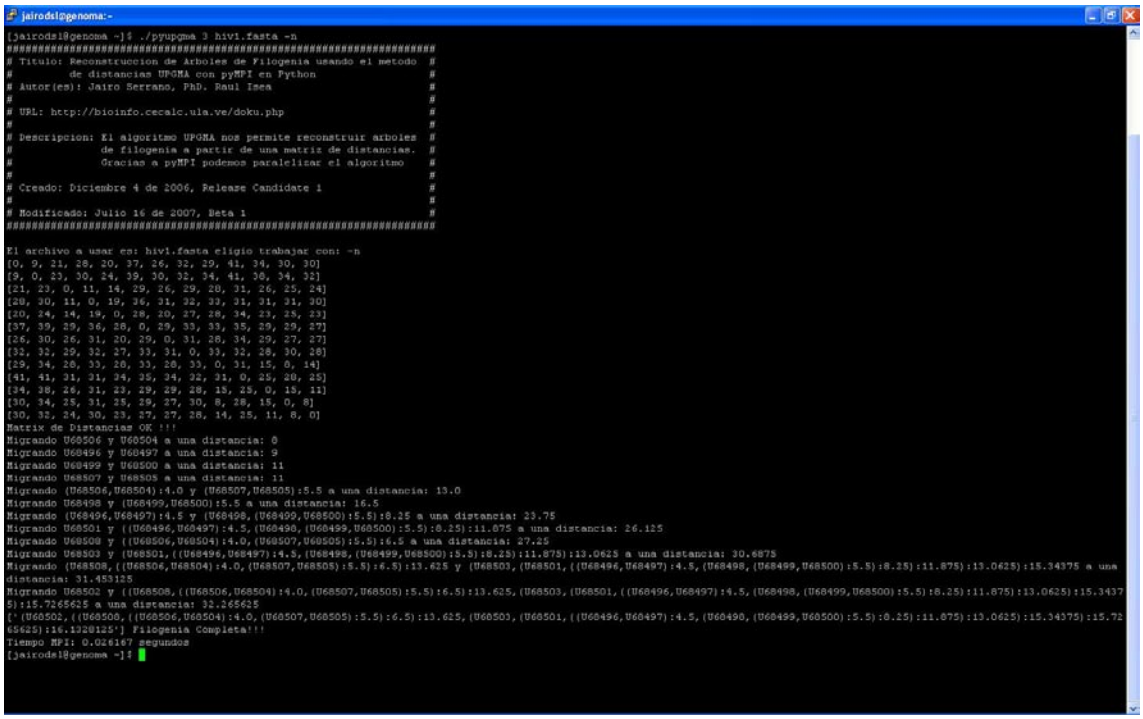


Fig. 6.7. pyUPGMA ejecutandose en tres procesadores para determinar la filogenia del HIV-1.

Lo anterior demuestra que a medida que aumente el número de procesadores aumentará el tiempo de respuesta, por lo que para hallar la filogenia del HIV-1 el mejor tiempo de respuesta se logró con tres procesadores. Dicho tiempo en el cálculo es muy similar que al calcularlo en forma secuencial.

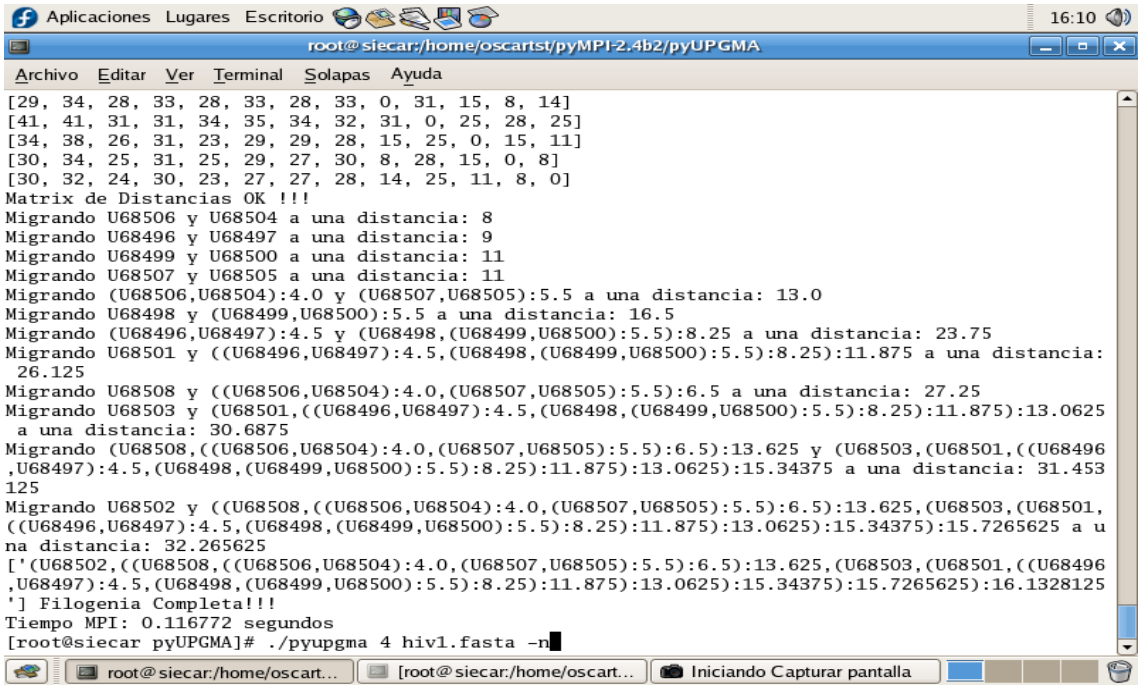


Fig. 6.8. pyUPGMA ejecutandose en cuatro procesadores. El tiempo de respuesta se incremento en casi una decima de segundo.

6.2.3. Árbol Filogenético del HIV-1 obtenido con el programa Jalview

Jalview[14] es una herramienta escrita en Java para análisis y edición de alineamientos de secuencias. Se emplea Jalview para comparar el árbol filogenético resultante con pyUPGMA+Treeview, ya que puede usar UPGMA para reconstruir arboles filogenéticos, pero como todos los programas[15] disponibles para determinar filogenia, Jalview agrega o modifica la manera como determina la matriz de distancias: un porcentaje de identidad entre dos secuencias en cada posición alineada es calculado para construir la matriz de distancias.

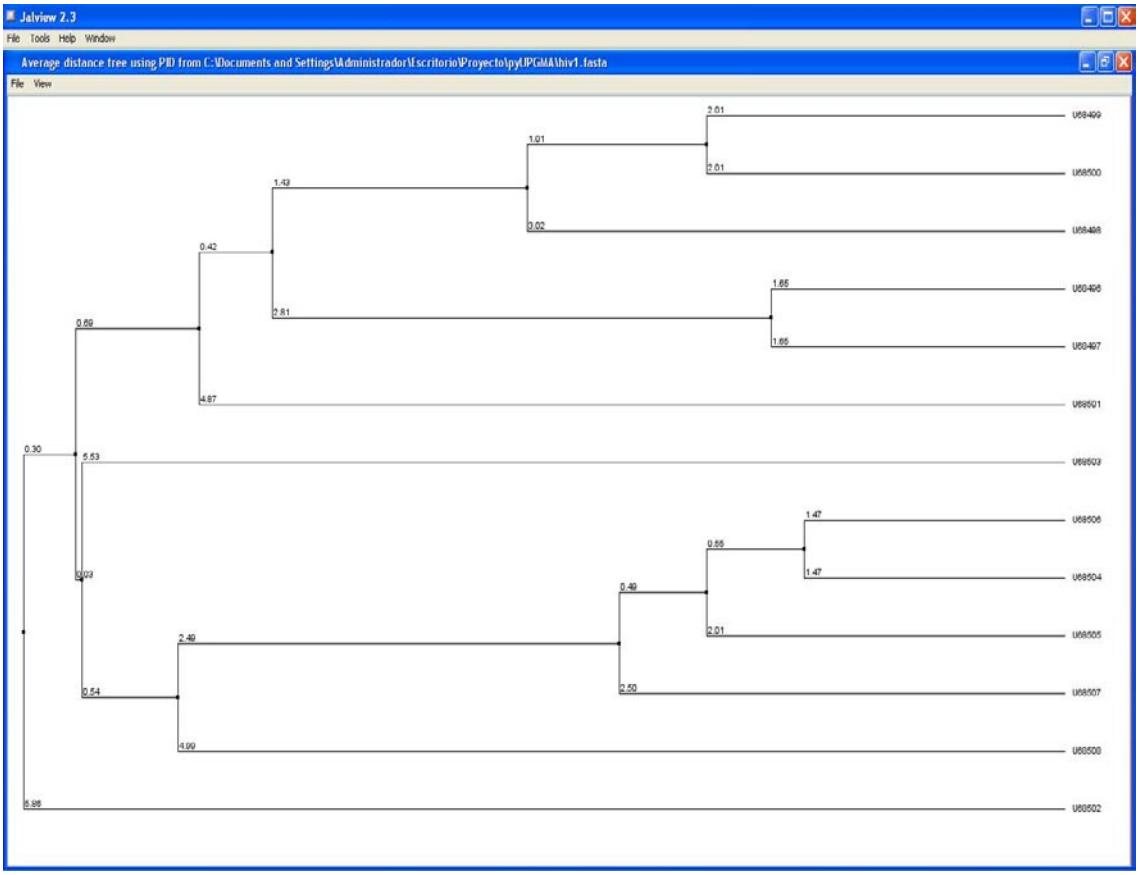


Fig. 6.9. Arbol Filogenético del HIV-1 obtenido con Jalview.

El árbol obtenido con pyUPGMA+Treeview es muy similar al mostrado por Jalview a pesar de no usar el mismo método para determinar la matriz de distancias. Por lo que se muestra la validez del algoritmo desarrollado en este proyecto.

---

[14] Clamp, M., Cuff, J., Searle, S. M. and Barton, G. J. (2004), "The Jalview Java Alignment Editor", Bioinformatics, 20, 426-7.

[15] http://evolution.genetics.washington.edu/phylip/software.html#Distance

6.3. PRUEBA DEL HPV

En esta prueba se usaron 105 secuencias del Virus del Papiloma Humano tomadas de NCBI[16] usando como palabra clave “papillomavirus”, las cuales se almacenaron en el archivo “papiloma.fasta” en formato fasta. Realmente la idea de este segundo caso, es determinar por medio del modelo si se encuentró una disminución entre los tiempos de ejecución secuencial y paralelo.

6.3.1. HPV calculado con la versión de pyUPGMA en secuencial

De la misma forma que se evaluó el rendimiento de pyUPGMA secuencial con la prueba del HIV-1 se procede con las secuencias del virus del Papiloma Humano.

© *Librería Time de Python:*

Los tiempos obtenidos para el calculo de la filogenia del papilloma virus en la versión secuencial de pyUPGMA oscilaron entre **2.25 segundos** y **2.46 segundos**. Tiempos que comparados con los obtenidos para las 13 secuencias del HIV-1 parecen mucho más grandes ya que para el papilloma virus se emplearon 105 secuencias, pero que son relativamente pequeños, si se mide la longitud de cada secuencia. Cabe aclarar que en la prueba con el HIV-1 no habíamos tenido en cuenta que el cluster no esta dedicado solo para pyUPGMA, otras aplicaciones se ejecutan constantemente y pueden interferir en el rendimiento de las operaciones.

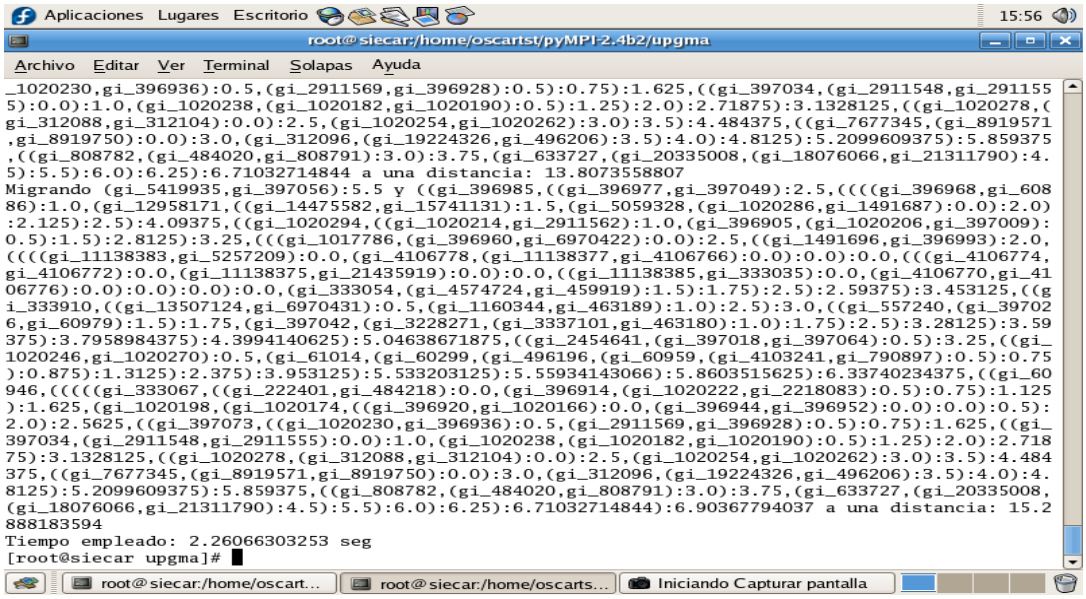


Fig. 6.10. Filogenia del papiloma virus obtenida en 2.260 segundos con pyUPGMA

[16] The National Center for Biotechnology Information (NCBI) is part of the United States National Library of Medicine (NLM), a branch of the National Institutes of Health.

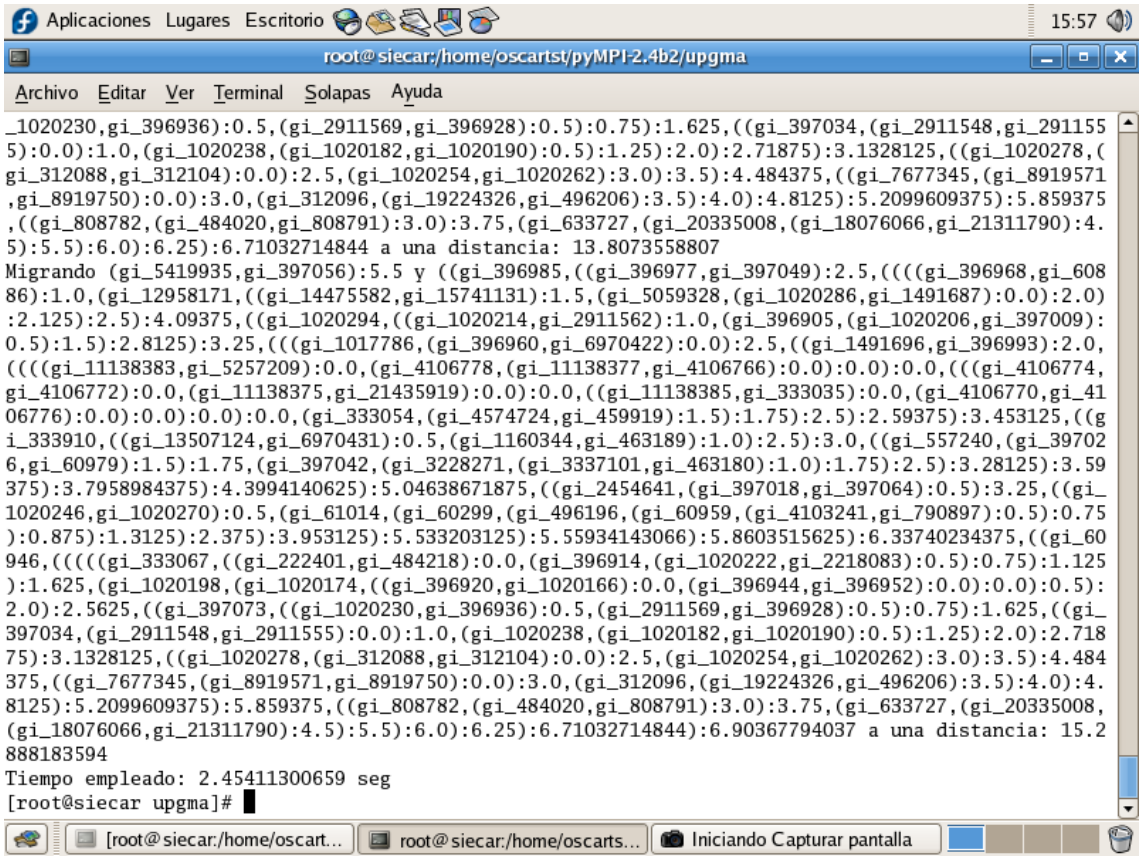


Fig. 6.11. Filogenia del papiloma virus obtenida en 2.454113 segundos obtenida con pyUPGMA

Por último se visualiza el árbol filogenético en la figura 6.11 a partir del archivo llamado “Filo_papiloma.fasta.newick” (obtenido al ejecutar pyUPGMA con las secuencias papiloma.fasta) en el programa Treeview. De la misma forma que para HIV-1 el archivo obtenido con la filogenia para el HPV fue el mismo en los prototipos secuencial tanto paralelo.

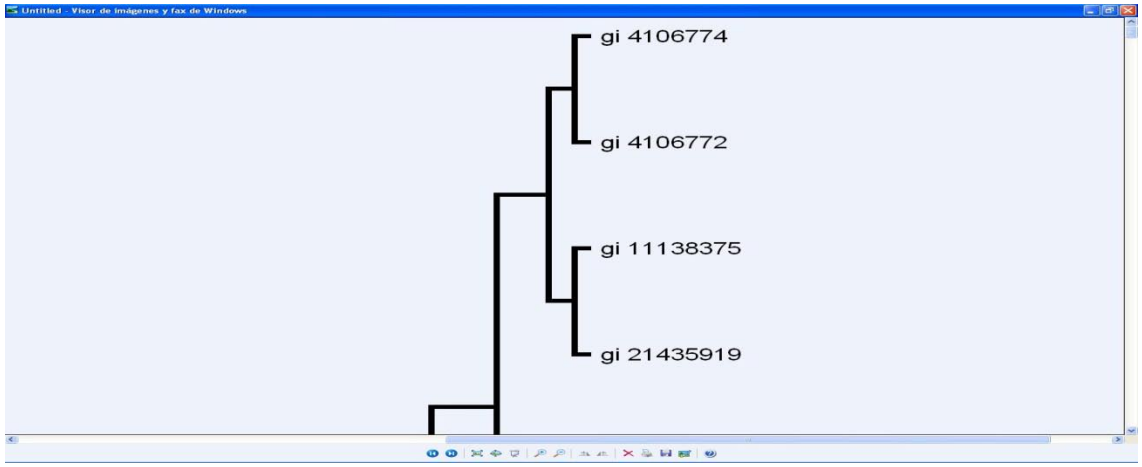


Fig. 6.12. Arbol Filogenético del HPV obtenido con Treeview: Extremo superior de la primera hoja de ocho hojas. (Ver anexo. Del árbol completo).

6.3.2. HPV calculado con la versión de pyUPGMA en paralelo

La misma restricción se hace necesaria para la prueba de HPV con pyUPGMA paralelo, al menos debe usarse tres procesadores, un maestro y dos esclavos, como se indico en la sección 6.2.2.

© *Función mpi.time() de pyMPI:*

Se puede observar que para tres procesadores el tiempo empleado para determinar la filogenia de papiloma virus fue de **1.573 segundos** que comparado con los 2.355 segundos en promedio del resultado secuencial la diferencia de cálculo es aproximadamente **0.8 segundos** lo que es un 33% más rápido con respecto al secuencial**,** aunque es un tiempo menor que el secuencial no es una diferencia considerable. Incluso para cuatro procesadores el tiempo obtenido fue menor que el secuencial, hablamos de **1.778 segundos** lo que es un 24.5% más rápido con respecto al secuencial, pero que lo que nos demuestra es que si seguimos aumentando el número de procesadores el tiempo de respuesta aumentara hasta alcanzar el tiempo secuencial (como se ha indicado en la sección 6.2.2)

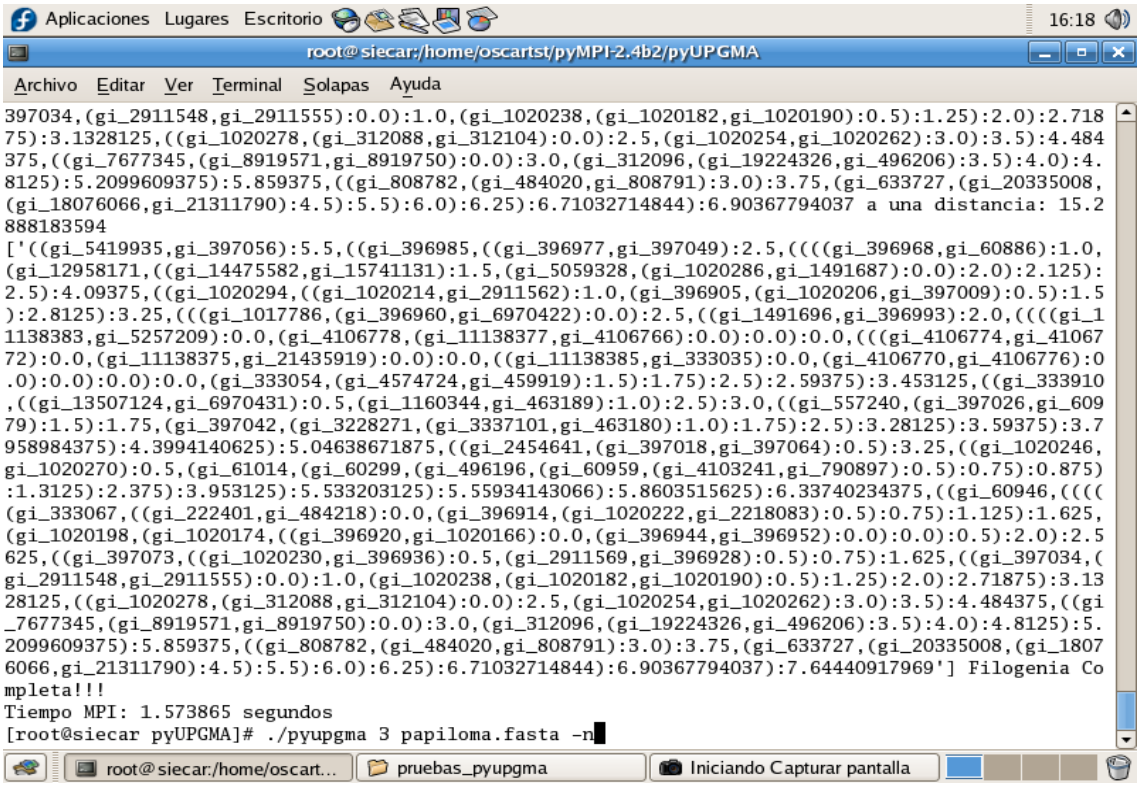


Fig. 6.13.Filogenia del papiloma virus obtenida en 1.573 segundos para tres procesadores con el programa pyUPGMA

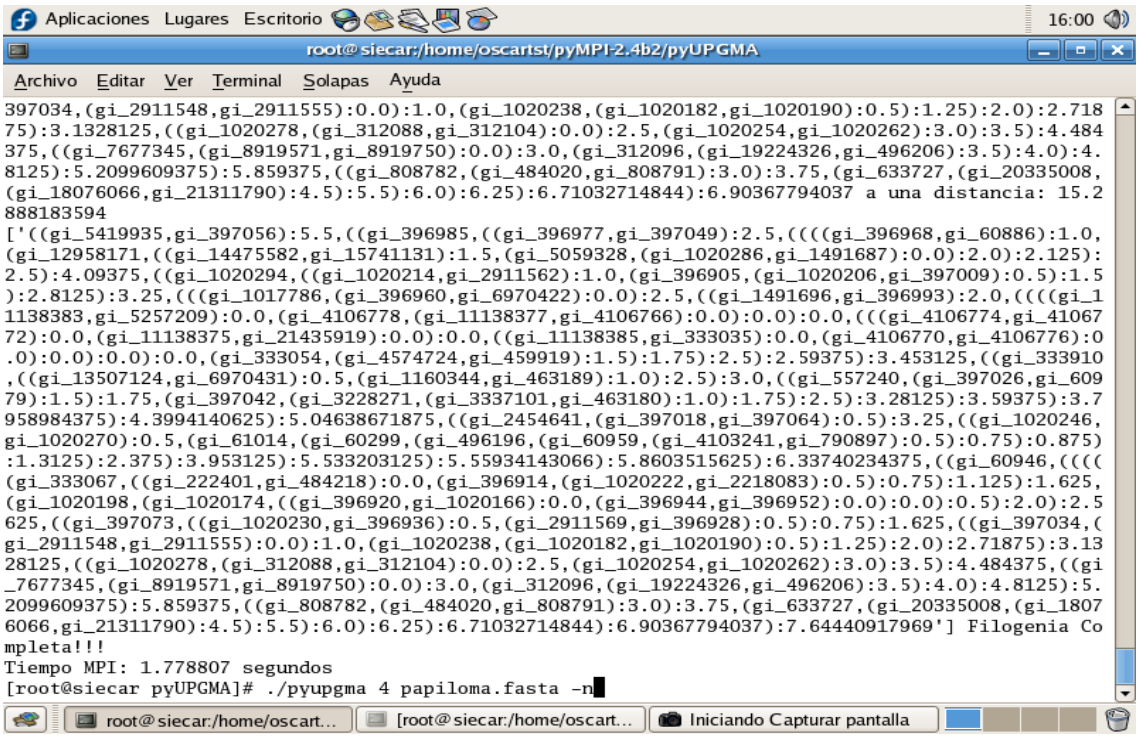


Fig. 6.14. Filogenia del papiloma virus obtenida en 1.778 segundo para cuatro procesadores con el programa pyUPGMA

6.3.3. Árbol Filogenético del HPV obtenido con el programa Jalview

El árbol obtenido con Jalview en el caso de las secuencias del virus del papiloma humano se muestra en la figura 6.15.

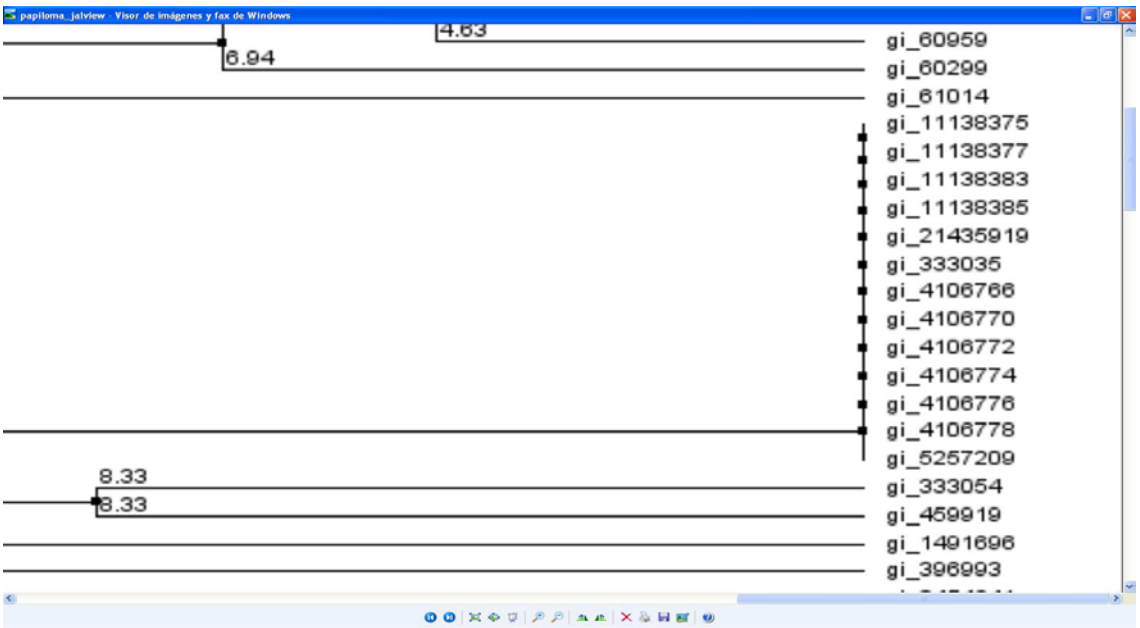


Fig. 6.15. Arbol Filogenético del HPV con Jalview. Parte intermedia del arbol. (Ver anexo. D para mejor detalle y resolución)

Entre el árbol filogenético obtenido con Jalview para papiloma virus y el obtenido con pyUPGMA+Treeview nuevamente se observa gran similaridad en varios rangos de secuencias (por ejemplo: las secuencias gi_11138375 a la gi_5257209), precisamente por la manera en que se construye la matriz de distancias en Jalview, por la cantidad de secuencias de HPV y la longitud de las mismas.

7. CONCLUSIONES

- © El programa implementado en esta tesis llamado pyUPGMA es la primera versión paralela Open Source del algoritmo para cálculo en filogenia.

- © Se implementó el algoritmo UPGMA en el lenguaje de programación Python tanto para ejecución en secuencial como en paralelo (dicha aplicación se llamó pyUPGMA) .

- © Los resultados obtenidos son los esperados si dichos cálculos se realizaran en un proceso manual a un grupo de secuencias de aminoácidos o nucleótidos para calcular su filogenia. Como se realizó en el caso de la Prueba Basica.

- © La librería Profile de Python proveyó información deterministica importante para discernir que rutinas se deben paralelizar en el código desarrollado en esta tesis. Dicha labor fue posible gracias la implementación que se desarrollo en la versión secuencial, que permite realizar un análisis de eficiencia y rendimiento para su posterior implementación en paralelo.

- © La aplicación desarrollada a lo largo de este trabajo interpreta los datos en formato FASTA y devuelve el resultado de la filogenia calculada en formato NEWICK, el cual es compatible con la mayoría de aplicaciones de cálculo y visualización de filogenia como fue Treeview al utilizar el formato de datos Standar.

- © Los arboles filogenéticos construidos a partir de pyUPGMA con Treeview se validaron con el programa Jalview, presentando la misma Topología.

- © pyUPGMA sobresale sobre la única versión paralela del algoritmo UPGMA no implementada, ya que pyUPGMA si considera el factor de comunicación entre procesadores.

- © La aceleración obtenida por pyUPGMA según la ley de Amdhal, demostró que el paradigma de memoria distribuida sobre el cual se diseñó e implementó no mejora sustancialmente el rendimiento, llegando a obtener tiempos de respuesta similares a los tiempos secuenciales.

8. RECOMENDACIONES

- Implementar una versión de pyUPGMA para arboles filogenéticos, dentro del mismo paradigma de memoria distribuida pero en arquitectura de 64bits.

- Desarrollar el programa pyUPGMA bajo el esquema de balanceo de carga entre los nodos esclavos del clúster.

- Incorporar a pyUPGMA el formato de datos de entrada XML, dado que esta siendo implementado como estándar para el manejo de la información en aplicaciones bioinformaticas.

- Desarrollar una interfaz web para el manejo del programa pyUPGMA como actualmente están implementando las aplicaciones bioinformáticas.

- Diseñar una interfaz gráfica para la visualización de los resultados obtenidos con el programa pyUPGMA en vez de emplear programas externos como Treeview.

BIBLIOGRAFIA

FILOGENIA Y BIOLOGIA EVOLUTIVA

- E.O Wiley; D. Siegel-Causey; D. R. Brooks; V. A. Funk. The Compleat Cladist “A primer of phylogenetic procedures”. Museum of Natural History, The University of Kansas. Kansas, 1991.

- FELSENSTEIN, Joseph. Theoretical Evolutionary Genetics. Department of Genome Sciencies and Department of Biology University of Washington. Washington, 2005.

- BALL, Harry. Phylogenetics Trees Made Easy: A How-to Manual. Sinauer Associates. 2004.

- GASCUEL, Oliver. Mathematics of Evolution and Phylogeny. Oxford University Press. 2005.

- ALBERT, Victor. Parsimony, Phylogeny and Genomics. Oxford University Press. 2005.

BIOINFORMATICA

- GIBAS, Cynthia. JAMBECK, Per. Developing Bioinformatics Computer Skills. OREILLY & Associates. 2001.

- BERGERON, Bryan. Bioinformatics Computing. Prentice Hall, Pearson Education. 2003.

- LESK, Arthur M. Introduction to Bioinformatics. OXFORD. 2002

PYTHON

- ZELLE, John. Python Programming: An introduction to computer science. Wartburg College Printing Services. 2002.

- NORTON, Peter; SAMUEL, Alex y otros. Beginning Python. Wiley Publishing, Inc. Indiana, 2005.

- BRUECK, Dave; TANNER, Stephen. Python Bible 2.1. Hungry Minds. New York, 2001.

- BEAZLY, David. Python Essential Reference 2nd Edition. New Riders Publishing. 2001

- MARTELLI, Alex; ASCHER, David y otros. Python Cookbook: Recipes from the Python Community. O´REILLY. 2005

- HETLAND, Magnus Lie. Beginning Python: From novice to professional. Apress. 2005

COMPUTACION DE ALTO RENDIMIENTO

- KARNIADAKIS, George y KIRBY, Robert. Parallel Scientific Computing in C++ and MPI. Ed. Cambridge University. 2001.

- BERTSEKAS, Dimitri. Parallel And Distributed Computation: Numerical Methods. Prentice-Hall. 1989.

- FOSTER, Ian. Designing and Building Parallel Programs, Addison-Wesley. 1995.

- GROPP, William. LUSK, Ewing. SKJELLUM, Anthony. Using MPI: Portable Parallel Programming with the Message-Passing Interface. 1994.

- DOWN, Kevin y SEVERANCE, Charles. High Performance Computing. Ed. O´really. 1998.

TRABAJOS DE GRADO

- BARRIOS, Carlos y CASALLAS, Juan Carlos. Desarrollo de aplicaciones para el procesamiento paralelo en una red de PCS. Universidad Industrial de Santander. 2002.

TECNICOS

- A, Gottlieb; K, Hwang; S, Sahni. Journal of Parallel and Distributed Computing. Elsevier. 2006.

- PRESSMAN, Roger. Ingeniería del Software: Un Enfoque Práctico. Ed. Mc Graw Hill. 2002.

- MILLER, P. Parallel, distributed scripting with Python. Center for Applied Scientific Computing Lawerence Livermore National Laboratory. 2002

- MILLER, P. pyMPI – An Introduction to parallel python using MPI. Center for Applied Scientific Computing Lawerence Livermore National Laboratory. 2002

Anexo A

Resumen y Poster presentado en el “International Workshop on Collaborative Bioinformatics RIBIO-EMBNET June 11-14th 2007” en Torremolinos, España.

Abstract

pyUPGMA: A Parallel Python Unweighted Pair Group Method with Arithmetic Mean Algortihm

Jairo Serrano, Universidad Industrial de Santander; Bucaramanga, Colombia.
Raúl Isea – CeCalcULA; Merida, Venezuela.

We introduce pyUPGMA, an implementation for phylogenetic analysis to reconstruct phylogenetic trees [1] based on the simplest method for their analyses, called Unweighted Pair Group Method with Arithmetic Mean – UPGMA[2]. Phylogeny is demanding computationally because the number of possible solutions increases quickly as taxa number also increases. [3]. The code is written 100% in Python[4] but we employed an extension that provides control over parallel codes through Message Passing Interface MPI[5] called pyMPI[6]. The reason to use pyMPI is twofold. Firstly, it is an open source extension for Python that allows the users to write parallel scripts very quickly without compilation, and secondly, pyMPI supports most APIs from MPI [7]. Finally, HIV-1 phylogenetic tree was reconstructed to know its origins[8].

[1]Methods for Phylogney Reconstruction, Ph.D. Antonio Barbadilla
[2] http://en.wikipedia.org/wiki/UPGMA
[3] http://es.wikipedia.org/wiki/Taxa
[4]Python, http://www.python.org
[5]MPI, http://www.mpi-forum.org/
[6]pyMPI, Library for Python, http://pympi.sourceforge.net/
[7] Parallel, Distributed Scripting with Python, P. Miller.
[8] http://www.mathworks.com/products/demos/bioinfo/hivdemo/hivdemo.html

Poster:

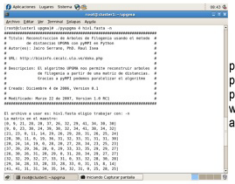
A Parallel Python Unweighted Pair Group Method with Arithmetic mean Algorithm

Serrano Latorre, Jairo ; Universidad Industrial de Santander - Colombia
Isea, Raúl; CeCalcULA - Venezuela

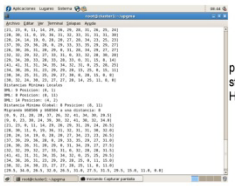
GETTING STARTED

pyUPGMA, is a 100% written implementation in Python with pyMPI extension, for phylogenetic analysis allowing reconstruction of phylogenetics trees [1], based onto Unweighted Pair Group Method with Arithmetic mean Algorithm. Many programs like Phylip, Paup, Mega and others, find out phylogeny based onto distances matrix, but they are not really clear to the user, they run sequentially, their results can not be use in different programs and they change the original UPGMA [2] algorithm.

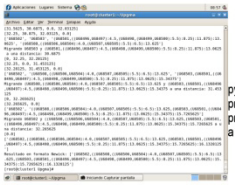
The aim of pyUPGMA was make a parallel version of UPGMA, searching improve the performance of sequential version of the original algorithm, actually don't exist a parallel version of UPGMA, PRAM computational model proposal has been stablished but it doesn't consider factor communication between processors.



pyUPGMA: receive parameters like number of processors and fasta file with sequences already aligned.



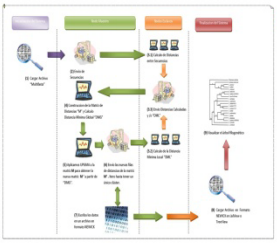
pyUPGMA: running step by step and parallell to find out HIV-1 phylogeny.



pyUPGMA: at the end of processing the results are printed in a NEWICK format in a file.

METHODOLOGY

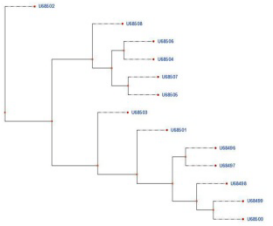
Designing and building of pyUPGMA is based onto Ian Foster´s methodology for designing and building parallel applications. In the case of pyUPGMA has been necessary combine Domain Decomposition to divide data (sequences) and Functional Decomposition to divide jobs (distances measure) throughout processors from the cluster. pyUPGMA run in two different clusters, Genoma the cluster from CeCalcULA with OS Rocks 4.2.1, and SALI a virtual cluster with OS Fedora 6. Phylogeny for **HIV-1** was calculated to test the functionality of pyUPGMA . Finally, Python and pyMPI are Open Source, and pyMPI support the most APIs from the standar MPI [3]. Python [4] version is 2.4 and pyMPI [5] version 2.4.b4.



Working Scheme of pyUPGMA.

RESULTS

Phylogeny obtained from pyUPGMA with TreeView [6] was used to reconstruct phylogenetic tree for HIV-1, it has a high degree of similarity with phylogenetic tree gotten through the popular program JalView [7]. The difference at UPGMA algorithm from Jalview is that it count the percentage of identity between two sequences in each aligned position. Additionally, the high dependency in data from distances matrix when parallelize this axis of UPGMA algorithm was tried, but punished the communication process between processors and favorable changes weren't in performance respect the sequential version of UPGMA.

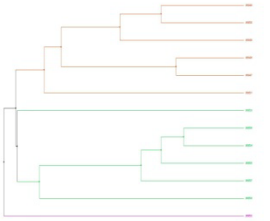


HIV-1 Phylogenetic Tree got with pyUPGMA with Treeview

Source Code from pyUPGMA.

CONCLUSIONS

Performance from pyUPGMA will be noticeably better if focus the calculus of phylogeny for consensus trees, because pyUPGMA is developed under paradigm of Distributed Memory, actually most of clusters works under it. If want improve the performance just for one phylogeny using UPGMA algorithm will need to break data dependency sharing variables in the same space memory throughout paradigm of Shared Memory and the standar OpenMP [8]. Finally the phylogeny got for HIV-1 is transparent for the user and clearly identifiable step by step when pyUPGMA is running.



HIV-1 Phylogenetic Tree got with Jalview

REFERENCES

[1] Methods for Phylogeny Reconstruction, Ph.D. Antonio Barbadilla
[2] Upgma Method, http://en.wikipedia.org/wiki/UPGMA
[3] Message Passing Interface, http://www.mpi-forum.org/
[4] Programming Language, http://www.python.org
[5] Parallel Library for Python, http://pympi.sourceforge.net/
[6] Treeiew, http://taxonomy.zoology.gla.ac.uk/rod/treeview.html
[7] Java alignment editor, http://www.jalview.org/
[8] Open Multi Processing, http://www.openmp.org

Acknowledgment:
To God, Our Families, Friends and the forgotten.

To whom it may concern:

This is to certify that

SERRANO LATORRE, JAIRO

has attended the

International Workshop on Collaborative Bioinformatics

jointly organized by

European Molecular Biology Network

and

Red Iberoamericana de Bioinformática

Torremolinos, Málaga, Spain

11-14 June 2007

José R. Valverde

Head of Scientific Computing Service

CNB/CSIC

Anexo B

Python

Tomado de: Guía de Aprendizaje de Python
(http://pyspanishdoc.sourceforge.net/tut/tut.html)

Python es un lenguaje de programación fácil de aprender. Tiene eficaces estructuras de datos de alto nivel y una solución de programación orientada a objetos simple pero eficaz. La elegante sintaxis de Python, su gestión de tipos dinámica y su naturaleza interpretada hacen de él el lenguaje ideal para scripts y desarrollo rápido de aplicaciones, en muchas áreas y en la mayoría de las plataformas.

El intérprete de Python y la extensa biblioteca estándar están disponible libremente, en forma de fuentes o ejecutables, para las plataformas más importantes en la página web de Python, http://www.python.org, y se pueden distribuir libremente. La misma página contiene también distribuciones y direcciones de muchos módulos, programas y herramientas Python de terceras partes, además de documentación adicional.

Si en alguna ocasión ha diseñado un guion o script para intérprete de órdenes (o *Shell script*) de Unix/Linux largo, puede que conozcas esta sensación: Te encantaría añadir una característica más, pero ya es tan lento, tan grande, tan complicado...O la característica involucra una llamada al sistema u otra función accesible sólo desde C. El problema en sí no suele ser tan complejo como para transformar el guion en un programa en C. Igual el programa requiere cadenas de longitud variable u otros tipos de datos (como listas ordenadas de nombres de fichero) fáciles en sh, pero tediosas en C. O quizá no tiene tanta soltura con C.

Otra situación: Quizá tengas que trabajar con bibliotecas C diversas y el ciclo normal C escribir-compilar-probar-recompilar es demasiado lento. Necesitas desarrollar software con más velocidad. Posiblemente has escrito un programa al que vendría bien un lenguaje de extensión y no quieres diseñar un lenguaje, escribir y depurar el intérprete y adosarlo a la aplicación.

En tales casos, Python puede ser el lenguaje que necesitas. Python es simple, pero es un lenguaje de programación real. Ofrece más apoyo e infraestructura para programas grandes que el intérprete de órdenes. Por otra parte, también ofrece mucha más comprobación de errores que C y, al ser un *lenguaje de muy alto nivel*, tiene incluidos tipos de datos de alto nivel, como matrices flexibles y diccionarios, que llevarían días de programación en C. Dados sus tipos de datos más generales, se puede aplicar a un rango de problemas más amplio que *Awk*[17] o incluso *Perl*[18], pero muchas cosas son, al menos, igual de fáciles en Python que en esos lenguajes.

Python permite dividir su programa en módulos reutilizables desde otros programas Python. Viene con una gran colección de módulos estándar que puedes utilizar como base de tus programas (o como ejemplos para empezar a aprender Python). También hay módulos incluidos que proporcionan E/S de ficheros, llamadas al sistema, sockets y hasta interfaces GUI (Graphical User Interface) como Tk.

[17] http://www.gnu.org/software/gawk/manual/gawk.html
[18] http://www.perl.org/

Python es un lenguaje interpretado, lo que ahorra un tiempo considerable en el desarrollo del programa, pues no es necesario compilar ni enlazar. El intérprete se puede utilizar de modo interactivo, lo que facilita experimentar con características del lenguaje, escribir programas desechables o probar funciones durante el desarrollo del programa de la base hacia arriba.

Python permite escribir programas muy compactos y legibles. Los programas escritos en Python son típicamente mucho más cortos que sus equivalentes en C o C++, por varios motivos:

- Los tipos de datos de alto nivel permiten expresar operaciones complejas en una sola sentencia.
- El agrupamiento de sentencias se realiza mediante sangrado (indentación) en lugar de begin/end o llaves.
- No es necesario declarar los argumentos ni las variables.

A propósito, el nombre del lenguaje viene del espectáculo de la BBC ``Monty Python's Flying Circus'' (el circo ambulante de Monty Python) y no tiene nada que ver con desagradables reptiles.

Introducción Informal a Python

- Comentarios

Los comentarios en Python empiezan por el carácter almohadilla, "#", y se extienden hasta el final de la línea física. Se puede iniciar un comentario al principio de una línea o tras espacio en blanco o código, pero no dentro de una constante literal, por ejemplo:

```
# éste es el primer comentario
fiambre = 1                 # y éste
                            # ... ¡y un tercero!
cadena = "# Esto no es un comentario."
```

- Números

El intérprete funciona como una simple calculadora: Tú tecleas una expresión y él muestra el resultado. La sintaxis de las expresiones es bastante intuitiva: Los operadores +, -, * y / funcionan como en otros lenguajes (p. ej. Pascal o C). Se puede usar paréntesis para agrupar operaciones. Por ejemplo:

```
>>> 2+2
4
>>> # Esto es un comentario
... 2+2
4
>>> 2+2  # un comentario junto al código
4
>>> (50-5*6)/4
5
>>> # La división entera redondea hacia abajo:
... 7/3
```

```
2
>>> 7/-3
-3
```

Al igual que en C, se usa el signo de igualdad "=" para asignar un valor a una variable. El valor de una asignación no se escribe:

```
>>> ancho = 20
>>> alto = 5*9
>>> ancho * alto
900
```

Se puede asignar un valor simultáneamente a varias variables:

```
>>> x = y = z = 0  # Poner a cero 'x', 'y' y 'z'
>>> x
0
>>> y
0
>>> z
0
```

La coma flotante funciona de la manera esperada. Los operadores con tipos mixtos convierten el operando entero a coma flotante:

```
>>> 4 * 2.5 / 3.3
3.0303030303
>>> 7.0 / 2
3.5
```

También funcionan de la manera esperada los números complejos: Los números imaginarios se escriben con el sufijo "`j`" o "`J`". Los números complejos con una parte real distinta de cero se escriben "`(`*real*+*imag*`j)`", y se pueden crear con la función "`complex(`*real*`,` *imag*`)`".

```
>>> 1j * 1J
(-1+0j)
>>> 1j * complex(0,1)
(-1+0j)
>>> 3+1j*3
(3+3j)
>>> (3+1j)*3
(9+3j)
>>> (1+2j)/(1+1j)
(1.5+0.5j)
```

Los números complejos siempre se representan como dos números de coma flotante, la parte real y la imaginaria. Para extraer una de las partes de un número complejo *z*, usa `z.real` y `z.imag`.

```
>>> a=1.5+0.5j
>>> a.real
1.5
```

```
>>> a.imag
0.5
```

Las funciones de conversión a coma flotante y a entero (float(), int() y long()) no funcionan con números complejos, pues no hay un modo único y correcto de convertir un complejo a real. Usa abs(z) para sacar su módulo (como flotante) o z.real para sacar su parte real.

```
>>> a=1.5+0.5j
>>> float(a)
Traceback (innermost last):
  File "<stdin>", line 1, in ?
TypeError: can't convert complex to float; use e.g. abs(z)
>>> a.real
1.5
>>> abs(a)
1.58113883008
```

En modo interactivo, la última expresión impresa se asigna a la variable _. Esto significa que, cuando se usa Python como calculadora, se facilita continuar los cálculos, por ejemplo:

```
>>> iva = 17.5 / 100
>>> precio = 3.50
>>> precio * iva
0.61249999999999993
>>> precio + _
4.1124999999999998
>>> round(_, 2)
4.1100000000000003
```

- Cadenas

Además de los números, Python también sabe manipular cadenas, que se pueden expresar de diversas maneras. Se pueden encerrar entre comillas simples o dobles:

```
>>> 'fiambre huevos'
'fiambre huevos'
>>> 'L\'Hospitalet'
"L'Hospitalet"
>>> "L'Hospitalet"
"L'Hospitalet"
>>> '"Sí," dijo.'
'"Sí," dijo.'
>>> "\"Sí,\" dijo."
'"Sí," dijo.'
>>> '"En L\'Hospitalet," dijo.'
'"En L\'Hospitalet," dijo.'
```

Las cadenas pueden ocupar varias líneas de diferentes maneras. Se puede impedir que el final de línea física se interprete como final de línea lógica mediante usando una barra invertida, por ejemplo:

```
hola = "Esto es un texto bastante largo que contiene\n\
varias líneas de texto, como si fuera C.\n\
```

```
    Observa que el espacio en blanco al principio de la línea es\
 significativo.\n"
print hola
```

mostraría lo siguiente:

```
Esto es un texto bastante largo que contiene
varias líneas de texto, como si fuera C.
         Observa que el espacio en blanco al principio de la línea
es significativo.
```

Se puede concatenar cadenas (pegarlas) con el operador + y repetirlas con *:

```
>>> palabra = 'Ayuda' + 'Z'
>>> palabra
'AyudaZ'
>>> '<' + palabra*5 + '>'
'<AyudaZAyudaZAyudaZAyudaZAyudaZ>'
```

Se puede indexar una cadena. Como en C, el primer carácter de una cadena tiene el índice 0. No hay un tipo carácter diferente; un carácter es una cadena de longitud uno. Como en Icon, las subcadenas se especifican mediante la *notación de corte*: dos índices separados por dos puntos.

```
>>> palabra[4]
'a'
>>> palabra[0:2]
'Ay'
>>> palabra[2:4]
'ud'
```

Crear una nueva cadena con el contenido combinado es fácil y eficaz:

```
>>> 'x' + palabra[1:]
'xyudaZ'
>>> 'Choof' + word[-1:]
'ChoofZ'
```

Los índices de corte tienen valores por omisión muy prácticos; si se omite el primer índice se supone cero y si se omite el segundo se supone el tamaño de la cadena sometida al corte.

```
>>> palabra[:2]    # Los primeros dos caracteres
'Ay'
>>> palabra[2:]    # Todos menos los primeros dos caracteres
'daZ'
```

Los índices pueden ser negativos, para hacer que la cuenta comience por el final. Por ejemplo:

```
>>> palabra[-1]     # El último carácter
'Z'
>>> palabra[-2]     # El penúltimo carácter
```

```
'a'
>>> palabra[-2:]    # Los dos últimos caracteres
'aZ'
>>> palabra[:-2]    # Todos menos los dos últimos
'Ayud'
```

- Listas

Python utiliza varios tipos de datos *compuestos*, que se utilizan para agrupar otros valores. El más versátil es la *lista*, que se puede escribir como una lista de valores (elementos) separada por comas entre corchetes. Los elementos de una lista no tienen que ser todos del mismo tipo.

```
>>> a = ['fiambre', 'huevos', 100, 1234]
>>> a
['fiambre', 'huevos', 100, 1234]
```

Como los índices de las cadenas, los índices de una lista empiezan en cero. Las listas también se pueden cortar, concatenar, etc.:

```
>>> a[0]
'fiambre'
>>> a[3]
1234
>>> a[-2]
100
>>> a[1:-1]
['huevos', 100]
>>> a[:2] + ['bacon', 2*2]
['fiambre', 'huevos', 'bacon', 4]
>>> 3*a[:3] + ['¡Hey!']
['fiambre', 'huevos', 100, 'fiambre', 'huevos', 100, 'fiambre',
'huevos', 100, '¡Hey!']
```

A diferencia de las cadenas, que son *inmutables*, es posible cambiar los elementos de una lista:

```
>>> a
['fiambre', 'huevos', 100, 1234]
>>> a[2] = a[2] + 23
>>> a
['fiambre', 'huevos', 123, 1234]
```

Es posible anidar listas (crear listas que contienen otras listas), por ejemplo:

```
>>> q = [2, 3]
>>> p = [1, q, 4]
>>> len(p)
3
>>> p[1]
[2, 3]
>>> p[1][0]
2
>>> p[1].append('xtra')     # Consulte la sección 5.1
```

```
>>> p
[1, [2, 3, 'xtra'], 4]
>>> q
[2, 3, 'xtra']
```

- Primeros pasos programando

Por supuesto, se puede usar Python para tareas más complejas que sumar y/o restar dos números. Por ejemplo, podemos escribir una secuencia parcial de la serie de Fibonacci de este modo:

```
>>> # Serie de Fibonacci:
... # La suma de dos elementos nos da el siguiente
... a, b = 0, 1
>>> while b < 10:
...       print b
...       a, b = b, a+b
...
1
1
2
3
5
8
```

Este ejemplo introduce varias características nuevas.

- La primera línea contiene una *asignación múltiple*: a las variables a y b se les asignan a la vez los nuevos valores 0 y 1. En la última línea se utiliza esto de nuevo, demostrando que las expresiones del lado derecho se evalúan antes que la primera de las asignaciones. Las expresiones del lado derecho se evalúan de izquierda a derecha.
- El bucle while se ejecuta mientras la condición (en este caso: b < 10) sea cierta. En Python, como en C, cualquier valor entero no cero es verdadero y 0 es falso. La condición también puede ser una lista o cualquier secuencia, cualquier cosa con longitud no cero es verdadero, las secuencias vacías son falso. La comprobación en este caso es simple. Los operadores de comparación estándar se escriben igual que en C: < (menor que), > (mayor que), == (igual a), <= (menor o igual a), >= (mayor o igual a) y != (diferente de).
- El *cuerpo* del bucle está *sangrado* (o indentado): Éste es el modo en que Python agrupa las sentencias. Python (todavía) no ofrece un servicio de edición de líneas sangradas, así que hay que teclear a mano un tabulador o espacio(s) para cada línea sangrada. En la práctica, los programas más complejos se realizan con un editor de texto y la mayoría ofrece un servicio de sangrado automático. Cuando se introduce una sentencia compuesta interactivamente, se debe dejar una línea en blanco para indicar el final (porque el analizador de sintaxis no puede adivinar cuándo has acabado) Observa que cada línea de un bloque básico debe estar sangrada al mismo nivel exactamente.
- La sentencia print escribe el valor de la expresión o expresiones dadas. Se diferencia de escribir simplemente la expresión (como hemos hecho antes en los ejemplos de la calculadora) en el modo en que gestiona las expresiones múltiples y las cadenas.

Anexo C

pyMPI

Tomado de: Parallel, Distributed Scripting with Python
(Pat Miller, Center for Applied Scientific Computing, Lawrence Livermore National Laboratory)

Los lenguajes interpretados son ideales para la construcción de herramientas de control y administración así como para proveer un marco de trabajo para componentes no críticos en el tiempo de grandes aplicaciones (ej. GUI y manipulación de archivos). En arquitecturas paralelas tales como SMP's y clúster, un lenguaje de scripting paralelo es necesario para proveer la misma funcionalidad. Como respuesta a la anterior necesidad surgió pyMPI un conjunto de mejoras paralelas para el lenguaje Python que permite a los programadores escribir scripts verdaderamente paralelos-distribuidos. pyMPI permite a los desarrolladores escribir módulos paralelos como una extensión de o para sus programas.

En lenguajes de alto nivel, C, C++, Fortran, la librería MPI da acceso a rutinas de distribución de datos. Pero se necesita más que eso. Muchas tareas razonables y comunes son mejor hechas en (o como una extensión para) lenguajes scripting. Considere herramientas de administración como password crackers, limpiador de archivos, etc…Estas tareas son simple de escribir en un lenguaje de scripting como Python (un intérprete de código abierto, portable y de libre distribución). Pero estas tareas piden ser hechas en paralelo.

pyMPI, es una implementación extendida de Python con una interfaz MPI. La herramienta hace muy fácil escribir scripts paralelos de Python para administración de sistemas, exploración de datos, post-procesamiento de archivos, incluso para verdaderas y completas simulaciones científicas. pyMPI permite también a los desarrolladores prototipar la distribución de datos para algoritmos paralelos en una manera fácil, intuitiva e interactiva, sin tener que compilar código, construir tipos MPI especializados, y mecanismos de serialización. pyMPI soporta la mayoría de las API de MPI. Permite el acceso a envíos, recepciones, barreras, mensajería asíncrona, comunicadores, peticiones y estados. En resumen, provee un completo ambiente funcional paralelo junto con un poderoso motor de scripting. Esta combinación simplifica la generación de herramientas distribuidas a gran escala para clúster.

Hay tres premisas básicas sobre las cuales pyMPI está construido:

- El scripting es más que solo diversión. Hay un número de aplicaciones para las cuales el scripting es la forma apropiada de resolver o por lo menos de prototipar la solución.
- MPI es la manera correcta de hacer paralelismo masivo (quizás no una buena manera, pero razonable de todos modos)
- Una disminución en el orden de magnitud en el tiempo de ejecución es aceptable por el ahorro humano del programador

Si se analizan las premisas anteriores tenemos que el scripting es más que solo una forma divertida de programar por que los lenguajes de scripting dan poder y flexibilidad al programador y pueden evitar un ciclo lento de compilar-enlazar-ejecutar-depurar. La programación paralela es dura y hacerla portable más dura aun. La librería MPI hace el problema al menos tratable. Nada es gratis, dado que los lenguajes de scripting tienden a ser uno o dos órdenes de magnitud más lentos que soluciones similares hechas puramente en lenguajes compilados.

Introducción a pyMPI

Una de las formas más simples de usar pyMPI es interactivamente, desde el prompt. Por lo general siempre se le pasara un nombre de archivo como parámetro a pyMPI y este además puede recibir otros parámetros

```
[root@cluster1 upgma]# mpirun –np 3 pyMPI
>>> import mpi
>>> mpi.rank
0
2
1

>>> print 'Ejecutandose',mpi.rank,'de',mpi.size
Ejecutándose 2 de 3
Ejecutándose 0 de 3
Ejecutándose 1 de 3


[root@cluster1 upgma]# mpirun –np 3 pyMPI holamundo.py
Hola mundo, soy el proceso 2 de 3
Hola mundo, soy el proceso 1 de 3
Hola mundo, soy el proceso 0 de 3
```

El código fuente de holamundo.py:

```
import mpi
print "Hola mundo, soy el proceso",mpi.rank,"de",mpi.size
```

Con la sentencia "import mpi" se carga el modulo mpi, que contiene las llamadas a la API de la librería de paso de mensajes. Los atributos "size" y "rank" del modulo mpi indica el numero de tareas cooperativas y el identificador único de la tarea respectivamente.

- *Operaciones Típicas con pyMPI*

MPI provee operaciones explicitas de comunicación punto a punto. Los mensajes pueden ser bloqueantes y no bloqueantes. Las primitivas son "send", "recv" y sendrecv. La forma más simple de "send" especifica un valor y un destino:

```
mpi.send(mensaje,destino)
```

En el lado de la recepción, el receptor puede definir:

```
mensaje, status = mpi.recv()
```

Status es una estructura que almacena el rank de la tarea que envió, y una etiqueta tag que por defecto esta en 0.

La programación paralela requiere coordinación de recursos. La forma más simple de alcanzar esto es través de una llamada a una barrera que actúa como un punto de encuentro. A continuación vemos como una barrera es usada para asegurar que todas las tareas han terminado el trabajo antes de declarar el trabajo "Hecho".

```
[root@cluster1 upgma]# mpirun –np 3 pyMPI barrera.py
```

```
Trabajo en 2
Trabajo en 0
Trabajo en 1
Hecho
```

Código fuente de barrera.py:

```
import mpi
data = open('foo%02d.data'%mpi.rank).read()
print 'Trabajo en',mpi.rank
DATA = data.upper()
open('foo%02d.DATA'%mpi.rank,'w').write(DATA)
mpi.barrier()
if mpi.rank == 0:
print 'Hecho'
```

El trabajo puede ser distribuido entre tareas usando un esquema de difusión. Por ejemplo

```
>>> lhs = mpi.bcast(rhs)
```

En la anterior sentencia, la difusión asegura que el valor de la variable "lhs" en todas las tareas cooperativas es el valor que la variable "rhs" tiene en la llamada tarea raíz. La tarea raíz es por defecto la única con rank 0. El valor de rhs puede ser cualquier tipo de dato Python, esto incluye tuplas, listas, diccionarios, instancias y otros. Tanto mpi.barrier y mpi.bcast son operaciones sincronizadas. Las tareas estarán bloqueadas hasta que cualquier otra tarea realice la operación.

Broadcast es usado para enviar información desde una tarea hacia todas las demás. La operación inversa se llama Reduction, la cual colecciona y procesa datos de muchas tareas. Como broadcast las reducciones son sincronizadas. Las reducciones requieren un valor para operar y una función para aplicar. pyMPI además permite usar funciones Python definidas por el usuario.

Consideremos un programa para integrar $\int_0^1 \frac{4}{1+x^2}$ (Note que $\frac{1}{1+x^2}$ define un cuarto de unidad de circulo en el cuadrante superior derecho, asi $\frac{4}{1+x^2}$ tiene la misma área que un circulo, π).

Podemos particionar el espacio entre un número pequeño de rectángulos sobre los cuales nosotros sumamos el área. La estrategia será que la tarea maestra (Rank 0) divulgue el número de rectángulos a las otras tareas. Cada tarea creara una suma local de sus áreas. Entonces todas las tareas contribuirán su suma local a la suma global (lo cual sería aproximadamente π).

```
import mpi
import string
import sys

def f(x): return 4.0/(1.0+x*x)

if mpi.rank == 0:
      n = string.atoi(sys.argv[1])
      mpi.bcast(n)
```

```
else:
      n = mpi.bcast()

h = 1.0/n
local_sum = 0.0

for i in range(mpi.rank+1,n+1,mpi.size):
      x = h*(i-0.5)
      y = f(x)
      local_sum += y

global_sum = mpi.reduce(local_sum,mpi.SUM)

if mpi.rank == 0:
      print 'PI es aproximadamente ',h*global_sum
```

Analizando, primero se define una pequeña función que implementa la función a integrar:

```
def f(x): return 4.0/(1.0+x*x)
```

Luego hay que hacer que el maestro obtenga el número de rectángulos de la línea de comandos:

```
n = string.atoi(sys.argv[1])
```

donde cada rectángulo será $h=\frac{1}{n}$ unidades de ancho. Se computan las sumas locales en cada uno de los nodos:

```
for i in range(mpi.rank+1,n+1,mpi.size):
      x = h*(i-0.5)
      y = f(x)
      local_sum += y
```

Para finalmente aplicar la operación de reducción mpi.SUM a las sumas locales y obtener la suma global:

```
global_sum = mpi.reduce(local_sum,mpi.SUM)
```

Anexo D

Guía de Uso de pyUPGMA

pyUPGMA es una aplicación que se ejecuta desde la consola de comandos de Linux, su ejecución puede hacerse de dos formas, la primera invocando directamente el entorno de computación paralela o a través de un script Shell que nos reduce el camino. Veamos los ejemplos.

1) [usuario@cluster ~]$ /opt/mpich/gnu/bin/mpirun –np <par1> /share/apps/python/pyMPI/bin/pyMPI pyUPGMA <par2> <par3> <par4>

El ejemplo del literal (1) esta diseñado para correr específicamente sobre el sistema operativo Linux Rocks 4.2.1 , si se quisiera instalar en otro clúster con otra distribución de Linux diferentes, sería cuestión de cambiar las rutas de las aplicaciones de MPI y pyMPI respectivamente.

Parámetros:

<par1>: Es el número de procesadores que se van a involucrar en el cálculo, es un valor entero entre 1 y el numero de nodos disponibles en el clúster.

<par2>: Es el nombre del archivo en formato FASTA que contiene las secuencias a las cuales se les desea encontrar su filogenia.

<par3>: Es el tipo de secuencia a analizar, aminoácidos o nucleótidos. Puede tomar como valores “-a” o “-A” para aminoácidos, “-n” o “-N” para nucleótidos.

<par4>: Es el número de procesadores que se van a involucrar en el cálculo. Debe tener el mismo valor de <par1>.

Por ejemplo, para calcular la filogenia de HPV con cuatro procesadores, el archivo a usar se llama papiloma.fasta, el tipo de secuencia será aminoácidos, se realiza de la siguiente forma:

```
[usuario@cluster ~]$ /opt/mpich/gnu/bin/mpirun –np 4
/share/apps/python/pyMPI/bin/pyMPI pyUPGMA papiloma.fasta -a 4
```

2) [usuario@cluster ~]$ **./**pyUPGMA <$1> <$2> <$3>

En esta segunda forma de ejecución hacemos uso de un script Shell para evitar tener que escribir rutas tan largas para invocar el entorno de computación paralela. El contenido del script es el siguiente:

```
#!/bin/bash
#
/opt/mpich/gnu/bin/mpirun –np $1 /share/apps/python/pyMPI/bin/pyMPI pyUPGMA $2 $3
$1
```

Parámetros:

<$1> : Es el número de procesadores que se van a involucrar en el cálculo, es un valor entero entre 1 y el numero de nodos disponibles en el clúster.

<$2>: Es el nombre del archivo en formato FASTA que contiene las secuencias a las cuales se les desea encontrar su filogenia.

<$3>: Es el tipo de secuencia a analizar, aminoácidos o nucleótidos. Puede tomar como valores “-a” o “-A” para aminoácidos, “-n” o “-N” para nucleótidos.

Por ejemplo, para calcular la filogenia de HIV-1 con cuatro procesadores, el archivo a usar se llama hiv1.fasta, el tipo de secuencia será nucleótidos, se realiza de la siguiente forma:

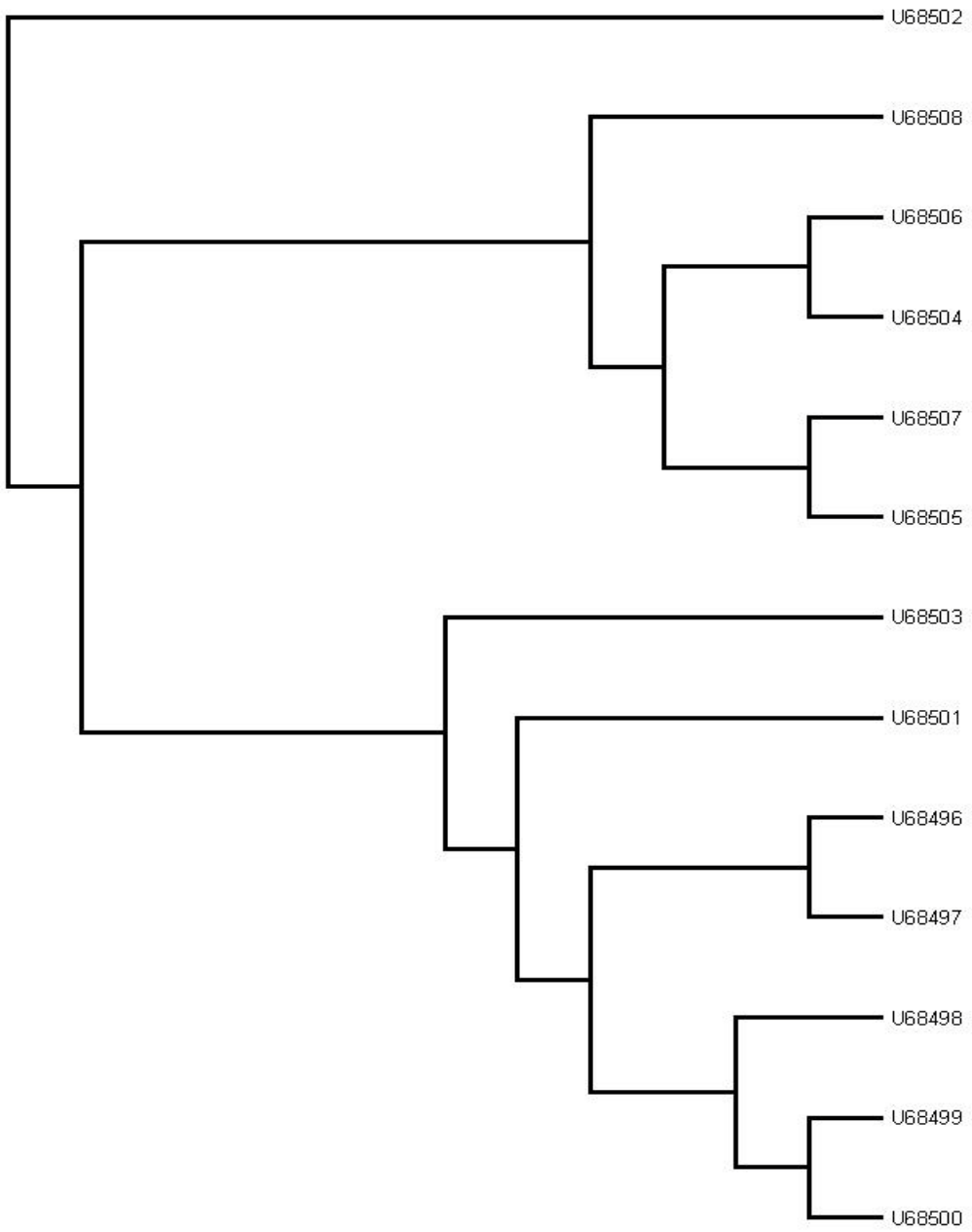
```
[usuario@cluster ~]$ ./pyUPGMA 4 hiv1.fasta –N
```

En este caso el parámetro <$1> se repite automáticamente, por lo cual no se necesita especificar un parámetro <$4>, a diferencia de la primera forma de ejecución donde si es necesario especificar este valor nuevamente.

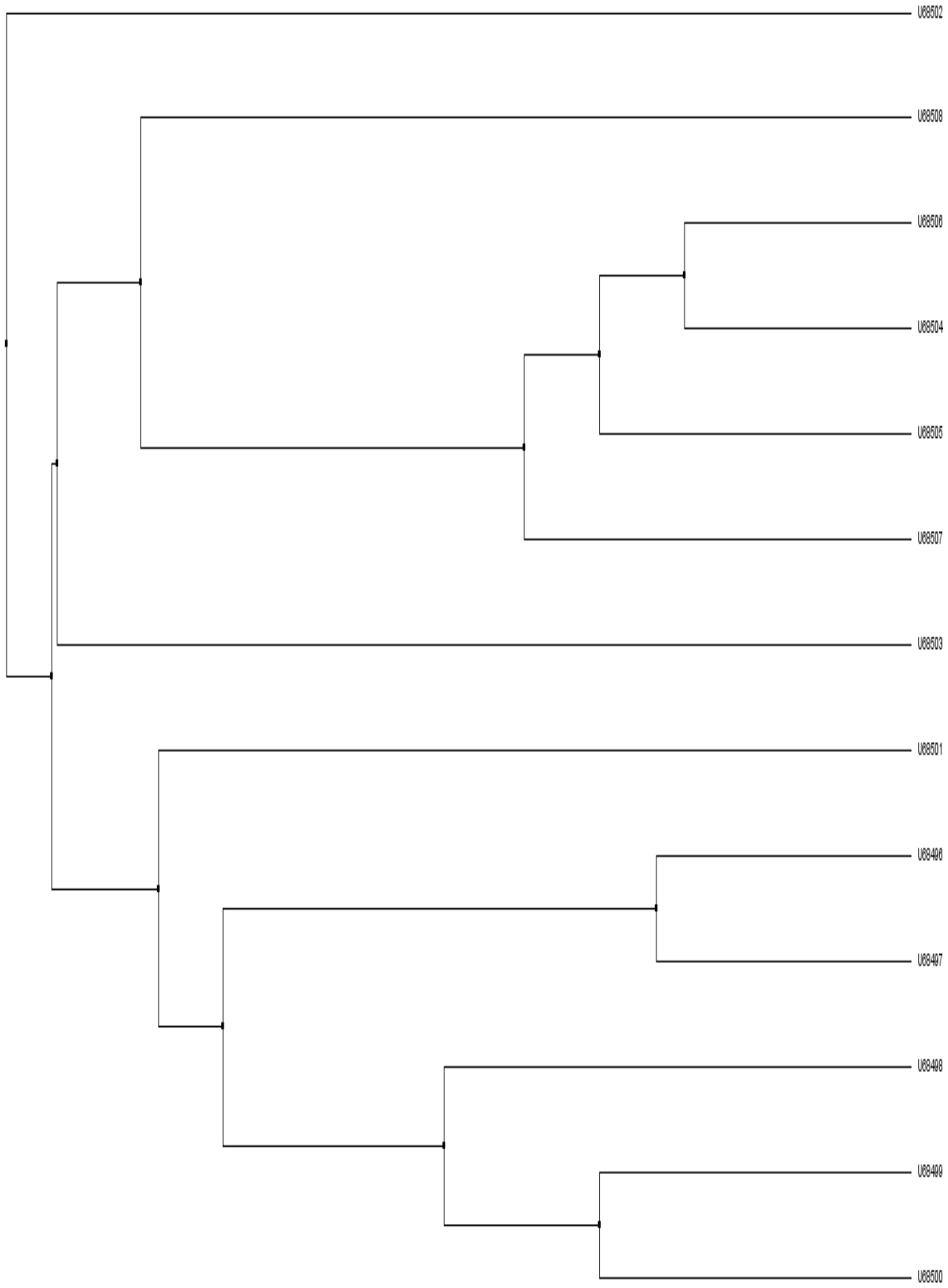
Anexo E

Arboles Filogenéticos del HIV-1 y HPV

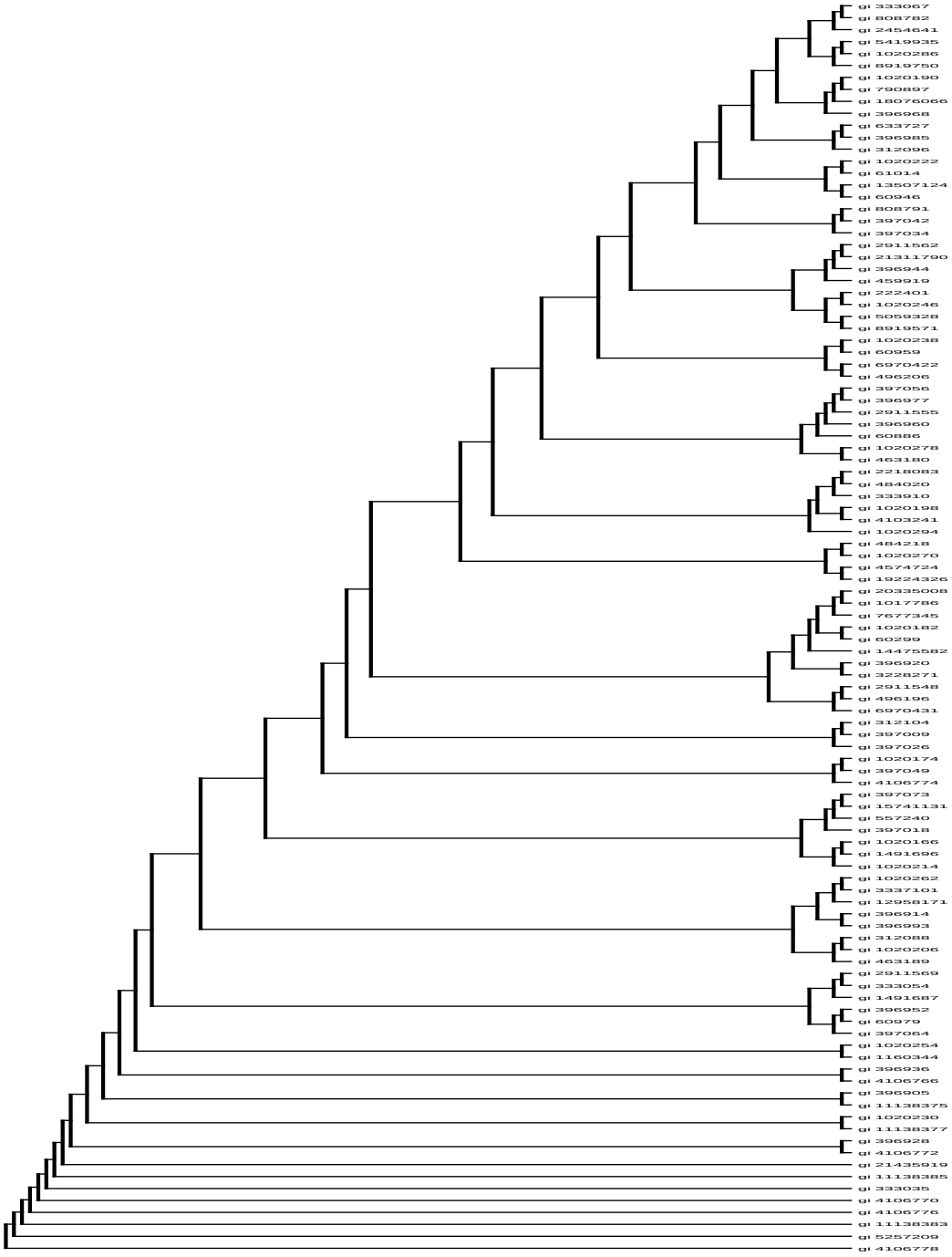
ÁRBOL FILOGENÉTICO DEL HIV-1 OBTENIDO CON TREEVIEW



ÁRBOL FILOGENÉTICO DEL HIV-1 OBTENIDO CON JALVIEW



ÁRBOL FILOGENÉTICO DEL HPV OBTENIDO CON TREEVIEW



ÁRBOL FILOGENÉTICO DEL HPV OBTENIDO CON JALVIEW