View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by Servicio de Difusión de la Creación Intelectual

Computación eficiente del alineamiento de secuencias de ADN sobre cluster de multicores

Enzo Rucci

Facultad de Informática

Universidad Nacional de La Plata

Director: Ing. Armando De Giusti

Co-Director: Dr. Marcelo Naiouf

Trabajo Final presentado para obtener el grado de

Especialista en Cómputo de Altas Prestaciones y Tecnología GRID

Abril 2013

Objetivo

El objetivo general de este trabajo consiste en presentar la aceleración en el tiempo de ejecución que se obtiene al paralelizar el algoritmo Smith-Waterman para el alineamiento de secuencias de ADN, debido en gran parte al aprovechamiento de las características de hardware de las arquitecturas de clusters actuales.

Los temas a abordar comprenden el análisis del problema, el estudio de distintas implementaciones del algoritmo secuencial y de posibles optimizaciones al mismo, y el desarrollo de un algoritmo paralelo (utilizando la librería de programación para memoria distribuida MPI) que saque provecho de las características físicas del hardware subyacente.

El alineamiento de secuencias de ADN es, en parte, el centro de todas las operaciones y análisis en el área de la bioinformática, tanto para la búsqueda de patrones entre secuencias de aminoácidos y nucleótidos, como para la búsqueda de relaciones filogenéticas entre organismos. El algoritmo Smith-Waterman permite realizar alineamientos locales de secuencias. Sin embargo, en la práctica se emplean diversas heurísticas, debido a los requerimientos de procesamiento y memoria del algoritmo Smith-Waterman. Si bien son más rápidas, las heurísticas no garantizan que el alineamiento óptimo sea encontrado. Un algoritmo paralelo que explote las características de la arquitectura de soporte y de las herramientas de programación paralela actuales para la misma permitirá reducir el tiempo de ejecución del proceso de alineamiento sin perder precisión en los resultados.

Este trabajo pretende aportar un algoritmo paralelo para el alineamiento local de secuencias de ADN junto a un análisis de rendimiento del mismo sobre una arquitectura de cluster actual para diferentes configuraciones de parámetros del problema y del sistema.

Resumen

Una de las áreas de mayor interés y crecimiento en los últimos años dentro del procesamiento paralelo es la del tratamiento de grandes volúmenes de datos, tales como las secuencias de ADN. El tipo de procesamiento extensivo de comparación para analizar patrones genéticos requiere un esfuerzo importante en el desarrollo de algoritmos paralelos eficientes [1].

El alineamiento de secuencias de ADN representa una de las operaciones más importantes dentro de la bioinformática. En 1981, Smith y Waterman desarrollaron un método para el alineamiento local de secuencias [2]. Sin embargo, en la práctica se emplean diversas heurísticas en su lugar, debido a los requerimientos de procesamiento y de memoria del algoritmo Smith-Waterman. Si bien son más rápidas, las heurísticas no garantizan que el alineamiento óptimo sea encontrado. Es por ello que resulta interesante estudiar cómo aplicar la potencia de cómputo de plataformas paralelas actuales de manera de acelerar el proceso de alinear secuencias sin perder precisión en los resultados.

Los niveles insostenibles de generación de calor y consumo de energía que se presentan al escalar al máximo la velocidad de los procesadores mononúcleos motivaron el surgimiento de los procesadores de múltiples núcleos (*multicore*). Un procesador multicore integra dos o más núcleos computacionales dentro de un único chip y, si bien estos son más simples y menos veloces, al combinarlos permiten mejorar el rendimiento global del procesador y al mismo tiempo hacerlo más eficiente energéticamente [3, 4]. Al incorporar este tipo de procesadores a los clusters convencionales, se da origen a una arquitectura conocida como cluster de multicores, que combina memoria compartida y distribuida, y donde la comunicación entre las diferentes unidades de procesamiento resulta ser heterogénea.

En este trabajo se presenta un algoritmo paralelo distribuido para el alineamiento de secuencias de ADN basado en el método Smith-Waterman para ser ejecutado sobre las arquitecturas de cluster actuales. Además, se realiza un análisis de rendimiento del mismo. Por último, se presentan las conclusiones y las posibles líneas de trabajo

futuro.

Publicaciones relacionadas a este trabajo

- Enzo Rucci, Franco Chichizola, Marcelo Naiouf, Laura De Giusti, Armando De Giusti. “Parallel pipelines for DNA sequence alignment on a cluster of multicores. A comparison of communication models.”, en *Journal of Communication and Computer (JCC)*, vol. 9, California (EEUU): David Publishing Company, 2012, núm. 12, pp. 1364-1371. ISSN: 1548-7709.

- Enzo Rucci, Armando De Giusti, Franco Chichizola, Marcelo Naiouf, Laura De Giusti, “DNA Sequence Alignment: hybrid parallel programming on multicore cluster”, en *Recent Advances in Computers, Communications, Applied Social Science and Mathematics*, N. Mastorakis, V. Mladenov, B. Lepadatescu, H. Reza Karimi, C. G. Helmis (Editores), vol. 1, Barcelona (España): WSEAS Press, 2011, pp. 183-190.

- Enzo Rucci, Armando E. De Giusti, Franco Chichizola. "Parallel Smith-Waterman Algorithm for DNA Sequences Comparison on Different Cluster Architectures", en *Proceedings of the 2011 International Conference on Parallel and Distributed Processing Techniques and Applications*, H. R. Arabnia (Editor), M. Ito, K. Joe, H. Nishikawa, H. Ishii, F. G. Tinetti, A. M. G. Solo, G. A. Gravvanis (Editores asociados), vol. 1, Las Vegas (EEUU): CSREA Press, 2011, pp. 666-672.

- Enzo Rucci, Armando De Giusti, Franco Chichizola, Marcelo Naiouf, Laura De Giusti. “Comparison of Communication/Synchronization Models in Parallel Programming on Multi-Core Cluster”, en *Computer Science & Technology Series – XVI Argentine Congress of Computer Science Selected Papers*, G. Simari, H. Padovani (Editores), Morón (Argentina): Editorial Edulp, 2011, pp. 71-81.

Índice general

1. Alineamiento de secuencias de ADN mediante el método Smith-Waterman 1

1.1. Bioinformática 1

1.2. Secuencias de ADN 2

1.3. Alineamiento de secuencias de ADN 2

1.3.1. Un caso sencillo: comparación de dos secuencias 3

1.3.2. Subsecuencias 3

1.3.3. Identidad y similitud 4

1.3.4. Similitud local y global 5

1.4. Algoritmo Smith-Waterman 5

2. Arquitectura cluster de multicores 9

2.1. Clusters 9

2.2. Clusters de multicores 10

3. Modelo de programación pasaje de mensajes y estándar MPI 13

3.1. Modelo de programación pasaje de mensajes 13

3.2. Estándar MPI 14

3.2.1. Operaciones de comunicación no bloqueantes 14

3.2.2. Comunicadores 15

3.2.3. Operaciones de comunicación colectivas 15

4. Evaluación de sistemas paralelos 17

4.1. Concepto de sistema paralelo 17

4.2. Fuentes de overhead en programas paralelos 17

4.2.1. Interacción entre procesos 18

4.2.2. Ocio 18

4.2.3. Exceso de cómputo 18

4.3. Métricas de rendimiento para sistemas paralelos 19
4.3.1. Tiempo de ejecución 19
4.3.2. Speedup 19
4.3.3. Eficiencia 20
4.3.4. Escalabilidad 21

5. Algoritmos para el alineamiento de secuencias de ADN por el método Smith-Waterman 22
5.1. Algoritmo secuencial 22
5.2. Algoritmo secuencial por bloques 23
5.3. Algoritmo paralelo 26
5.3.1. Modelo para el cálculo del tamaño *TBB* óptimo 27

6. Trabajo experimental 30
6.1. Arquitectura de soporte 30
6.2. Algoritmos utilizados 30
6.3. Pruebas realizadas 31
6.3.1. Primera fase de prueba 31
6.3.2. Segunda fase de prueba 31

7. Resultados 33
7.1. Resultados de la primera fase de pruebas 33
7.2. Resultados de la segunda fase de pruebas 35

8. Conclusiones y líneas de trabajo futuro 42

A. Estándar Message-Passing Interface (MPI) 48
A.1. Funciones básicas 48
A.2. Operaciones de comunicación no bloqueantes 51
A.3. Operaciones de comunicación colectivas 52

B. Estimación de parámetros del modelo para el cálculo del tamaño *TBB* óptimo en la arquitectura de soporte 55
B.1. Parámetros t_s y t_b 55
B.2. Parámetro t_c 56

C. Trabajos relacionados 57

Capítulo 1

Alineamiento de secuencias de ADN mediante el método Smith-Waterman

En este Capítulo se describe el proceso de alineamiento de secuencias de ADN, poniendo especial énfasis en el método Smith-Waterman para lograrlo.

1.1. Bioinformática

Durante las últimas dos décadas, la biología molecular se ha visto revolucionada debido no sólo al desarrollo de técnicas de secuenciación de ADN cada vez más rápidas sino también al progreso de las tecnologías informáticas, las cuales permiten almacenar, manipular y analizar enormes cantidades de información en forma cada vez más eficaz. El término *bioinformática* se comenzó a utilizar a mediados de los años 80 para referirse a aquellas aplicaciones de computadora que resuelven problemas biológicos [1]. En la actualidad, se utilizan computadoras para recolectar, almacenar, analizar y combinar datos biológicos. Se puede decir que bioinformática es la aplicación de tecnología de computadoras al manejo y análisis de datos biológicos.

La bioinformática es un área de investigación interdisciplinaria que actúa como puente entre las ciencias biológicas y las ciencias computacionales. El objetivo final de la bioinformática es descubrir la información importante que se encuentra oculta dentro de enormes volúmenes de datos y obtener una visión más clara de la biología de los organismos [5].

1.2. Secuencias de ADN

Una secuencia de ADN es una sucesión de letras representando la estructura primaria de una molécula real o hipotética de ADN, con la capacidad de transportar información. Las posibles letras son A, C, G, y T, que simbolizan las cuatro subunidades de nucleótidos de una banda ADN - adenina, citosina, guanina, timina - que son bases covalentemente ligadas a cadenas fosfóricas. Normalmente las secuencias se presentan pegadas unas a las otras, sin espacios, como por ejemplo la secuencia $AAAGTCTGAC$.

Las secuencias pueden derivarse de material biológico de descarte a través del proceso de secuenciación de ADN. El mismo hace referencia a un conjunto de métodos y técnicas que permiten determinar el orden de los nucleótidos (A, C, G y T) en una molécula de ADN. El conocimiento de las secuencias de ADN se ha convertido en un recurso indispensable para áreas de investigación básica, como la biología, así como también para áreas de investigación aplicada, como la biotecnología y la biología forense. El desarrollo de la secuenciación de ADN ha acelerado significativamente la investigación y los descubrimientos en la biología. La velocidad de las técnicas de secuenciación actuales permite que proyectos a gran escala puedan ser llevados a cabo, como por ejemplo, el Proyecto Genoma Humano. Otros proyectos relacionados, a menudo gracias a la colaboración a nivel mundial de científicos, han generado la secuencia completa de muchos animales, plantas y microorganismos [1].

1.3. Alineamiento de secuencias de ADN

En parte, el centro de todas las operaciones y análisis en el área de la bioinformática, lo tiene el alineamiento de secuencias, tanto para búsqueda de patrones entre secuencias de aminoácidos y nucleótidos, como para la búsqueda de relaciones filogenéticas entre organismos. De esta manera el alineamiento de secuencias permite responder una serie de preguntas para la biología: ¿qué tan parecidas son dos secuencias ya conocidas? ¿a qué puede llegar a parecerse una secuencia desconocida? Encontrarles una respuesta no ha sido una tarea fácil y ha existido una gran brecha entre los conocimientos y volúmenes de datos adquiridos durante largos años de investigación, y la eficiencia con la que estos pueden ser analizados. Sin embargo en la actualidad con la ayuda de sofisticados algoritmos y el desarrollo del campo de la biología computacional se ha logrado acortar la diferencia mencionada.

```
Sin alinear                                   Alineado

Secuencia 1     A G G C T C C T T G          Secuencia 1     A G G C T C C T T G
                | | | | | |                                  | | | | |   | | | |
Secuencia 2     A G G C T C T T G            Secuencia 2     A G G C T - C T T G
```

Figura 1.1: Uso del carácter '-' para alinear dos secuencias. Las barras verticales denotan emparejamientos idénticos: seis en el primer alineamiento, nueve en el segundo.

1.3.1. Un caso sencillo: comparación de dos secuencias

Una aproximación simplista para alinear dos secuencias consiste en colocar una frente a la otra e insertar caracteres adicionales para ponerlas en un alineamiento vertical, tal como se muestra en la Figura 1.1.

El proceso de alineamiento puede medirse mediante dos parámetros: el número de huecos (*gaps*) introducidos y el número de desemparejamientos (*mismatches*) que se mantienen en el alineamiento. Una métrica relativa a tales parámetros representa la distancia entre dos secuencias (la conocida como distancia de edición). Existen diversas métricas, y diferentes implementaciones de algoritmos similares pueden emplear distintas medidas de distancia para calcular y puntuar alineamientos [1].

1.3.2. Subsecuencias

El ejemplo presentado en la Sección 1.3.1 resulta ser muy simple: las secuencias son muy cortas, tienen casi la misma longitud y son casi idénticas. La realidad indica que esto normalmente no ocurre. Se considera entonces un par de secuencias más realista, donde la secuencia A tiene una longitud de 300 residuos, la B contiene 900 residuos y la secuencia A es en su totalidad idéntica a una porción de la secuencia B (en este caso se dice que A es una subsecuencia de B). Para poder alinear A con B, se deben insertar todos los huecos que hagan falta, tal como se muestra en la Figura 1.2.

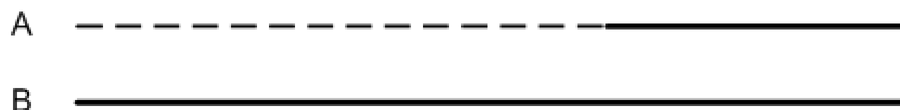

Figura 1.2: Alineamiento de secuencias A y B (siendo A idéntica a una parte de B).

El alineamiento presentado en la Figura 1.2 se puede hacer aun más realista. Para ello se considera que la secuencia A tiene 2 regiones extensas que muestran identidad

con la secuencia B, en lugar de ser una subsecuencia de la misma. En este caso, el procedimiento para realizar el alineamiento se ve modificado. En primer lugar, se deben identificar las regiones idénticas. Una vez identificadas, se deben insertar huecos en A para lograr el alineamiento con B, tal como se muestra en la Figura 1.3. Para finalizar el proceso, se debe encontrar la puntuación más alta entre las subsecuencias de A y B [1].

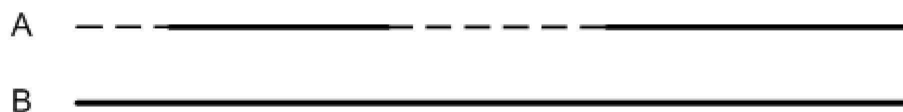

Figura 1.3: Alineamiento de secuencias A y B (siendo A idéntica a diferentes partes de B).

1.3.3. Identidad y similitud

El desarrollo de un programa que permita encontrar regiones de identidad estricta entre dos secuencias podría resolverse en tiempo y en forma razonables. Sin embargo, en general, el alineamiento no se restringe a emparejar subsecuencias, sino que implica la comparación de secuencias en toda su longitud. Un alineamiento completo debe determinar las posiciones de todos los residuos en ambas secuencias. Esto significa que es posible que muchos residuos tengan que situarse en posiciones que no son estrictamente idénticas. En ese caso la posición de los huecos en el alineamiento se hace más compleja de calcular. Una solución simple y rápida consiste en insertar huecos de forma no restringida, maximizando así el número de emparejamientos idénticos. Si bien esta solución permite alcanzar una puntuación óptima, el resultado de tal proceso no tendría sentido biológico. Es por ello que se introducen penalizaciones en la puntuación para minimizar el número de huecos que se inician (se abren) y a continuación se utilizan penalizaciones de extensión cuando el hueco debe ser extendido.

Realizar un alineamiento de secuencias por computadora significa generar un emparejamiento entre dos secuencias según un modelo matemático. El modelo describe, en términos generales, el concepto de alineamiento de dos secuencias y sus detalles: penalizaciones por huecos, impacto de las diferentes longitudes de las secuencias, efecto de la complejidad del alfabeto, entro otros. Estos se tratan mediante el uso de parámetros. Una elección adecuada de parámetros minimizará el número de huecos, mientras que su relajación permitirá, teóricamente, el alineamiento de cualquier par de secuencias arbitrarias. El hecho de que un programa produzca un alineamiento

de dos secuencias no debe tomarse como prueba, por sí mismo, de que exista una relación entre ellas [1].

1.3.4. Similitud local y global

Como se mencionó anteriormente, los alineamientos son modelos matemáticos cuyo comportamiento es susceptible de ser modificado mediante el uso de parámetros. Existen diferentes modelos, cada uno diseñado para recoger una variedad de características físicas de las secuencias biológicas, incluyendo, por ejemplo, su parentesco estructural, funcional o evolutivo. En este contexto, se debe, por lo tanto, tener en cuenta que no hay alineamientos correctos o incorrectos sino modelos distintos que reflejan diferentes perspectivas biológicas.

Generalmente, existen dos modelos que contemplan los alineamientos en forma bastante diferente. El primero considera la similitud en toda la extensión de las secuencias, por lo que es llamado alineamiento global. El segundo se centra en regiones de similitud sólo en partes de las secuencias, lo que se conoce como alineamiento local. Resulta importante comprender estas diferencias de manera de poder apreciar que las secuencias no son uniformemente semejantes y que, por lo tanto, no tiene sentido realizar un alineamiento global sobre secuencias que tienen sólo semejanzas locales.

La razón para buscar similitudes locales es que los sitios fundamentales (por ejemplo, sitios catalíticos de las enzimas) se localizan en regiones relativamente cortas, que se conservan con independencia de eliminaciones o mutaciones en las partes restantes de la secuencia. Por lo tanto, una búsqueda por similitud local puede producir resultados con más sentido y sensibilidad biológicos que una búsqueda que pretenda optimizar el alineamiento sobre la longitud completa de las secuencias [1].

1.4. Algoritmo Smith-Waterman

En 1981, Smith y Waterman describieron un método, conocido como el algoritmo Smith-Waterman, para encontrar regiones comunes de similitud [2]. El mismo consiste en un enfoque matricial y se emplea un proceso de retroceso para reconstruir los alineamientos con huecos. El método de Smith-Waterman ha servido como base para el desarrollo de otros algoritmos posteriores e incluso se lo utiliza como medida de referencia para diferentes técnicas de alineamiento [1].

Este método permite alinear dos secuencias de ADN insertando gaps (en caso

de ser necesario) con el fin de detectar regiones de similitud local que indiquen la presencia de una relación entre ambas secuencias, por medio de la asignación de un puntaje de similitud. Si es necesario abrir gaps, es decir no aparear ciertos elementos de las secuencias para lograr mejores alineamientos, se debe penalizar dicha acción. El algoritmo calcula el puntaje de similitud entre dos secuencias, y luego, de ser necesario, emplea un retroceso para construir el alineamiento óptimo [6].

A continuación se realiza una explicación del funcionamiento del algoritmo para encontrar el puntaje de similitud entre dos secuencias de ADN.

Dadas dos secuencias: $A = a_1 a_2 a_3 \ldots a_M$ y $B = b_1 b_2 b_3 \ldots b_N$, se construye una matriz H de *(N+1)*x*(M+1)*, de tal forma que las bases nucleótidos que forman la secuencia A etiquetan las filas (a partir de la 1) y los de B las columnas (a partir de la 1). A través de los siguientes pasos de calculan los valores de H que darán el puntaje de similitud entre A y B:

1. Inicializar en 0 la fila 0 y la columna 0 de H, como se indica en la Ecuación 1.1.

$$H_{i,0} = H_{0,j} = 0 \qquad para\, 0 \leq i \leq N\, y\, 0 \leq j \leq M \tag{1.1}$$

2. Calcular el valor de $H_{i,j} \forall i \in [1, ..., N]$ y $\forall j \in [1, ..., M]$ por medio de la Ecuación 1.2. Este valor indica la máxima similitud entre dos segmentos que terminan en a_i y b_j respectivamente.

$$H_{i,j} = max \begin{cases} 0 \\ H_{i-1,j-1} + V(a_i, b_j) \\ C_{i,j} \\ F_{i,j} \end{cases} \tag{1.2}$$

   a) $V(a_i, b_j)$ es la función de coincidencias que indica el puntaje dado por hacer coincidir a a_i y b_j. Se basa en una tabla de valores denominada *matriz de sustitución*, la cual describe la probabilidad de que una base nucleótida de la secuencias A en la posición i tenga ocurrencia en la secuencia B en la posición j. La matriz más común es aquella que premia con un valor positivo cuando a_i y b_j son idénticos y que penaliza con un valor negativo cuando no lo son.

   b) $C_{i,j}$ es el puntaje considerando un *gap* en la columna j, y se calcula por

medio de la Ecuación 1.3.

$$C_{i,j} = max_{1 \leq k \leq i} \{H_{i-k,j} - g(k)\} \tag{1.3}$$

c) $F_{i,j}$ es el puntaje considerando un *gap* en la fila i, y se calcula por medio de la Ecuación 1.4.

$$C_{i,j} = max_{1 \leq l \leq i} \{H_{i,j-l} - g(l)\} \tag{1.4}$$

d) *g(x)* es la función de penalización por un *gap* de longitud x, y se obtiene por medio de la Ecuación 1.5, siendo q la penalización por la inserción de un *gap* y r por la extension del mismo.

$$g(x) = q + rx \qquad (q \geq 0; r \geq 0) \tag{1.5}$$

3. Obtener el puntaje de similitud como se indica en la Ecuación 1.6.

$$G = max_{(0 \leq i \leq N; 0 \leq j \leq M)} \{H_{i,j}\} \tag{1.6}$$

4. Por último, si se desea obtener el par de segmentos con máxima similitud se realiza un proceso de retroceso (*backtracking*) a partir de la posición de la matriz H donde se encontró el valor G (que representa el final del alineamiento de mayor puntaje entre las dos secuencias) hasta llegar a una posición cuyo valor es 0, siendo éste punto el inicio del par de segmentos.

En la Figura 1.4 se muestra un ejemplo de la aplicación del algoritmo Smith-Waterman a dos secuencias $A = CAGCCUCGCUUAG$ y $B = AAUGCCAUUGACGG$, con los siguientes parámetros:

- $V(a_i, b_j) = 1 \; si \; a_i = b_j$.
- $V(a_i, b_j) = -1/3 \; si \; a_i \neq b_j$.
- $q = 1$.
- $r = 1/3$.

Por una cuestión de legibilidad, los valores de la matriz construida se redondean a un único decimal.

	x	C	A	G	C	C	U	C	G	C	U	U	A	G
x	0	0	0	0	0	0	0	0	0	0	0	0	0	0
A	0	1	1	0	0	0	0	0	0	0	0	0	1	0
A	0	0	1	0,7	0	0	0	0	0	0	0	0	1	0,7
U	0	0	0	0,7	0,3	0	1	0	0	0	1	1	0	0,7
G	0	0	0	**1**	0,3	0	0	0,7	1	0	0	0,7	0,7	1
C	0	1	0	0	**2**	1,3	0,3	1	0,3	2	0,7	0,3	0,3	0,3
C	0	1	0,7	0	1	**3**	1,7	1,3	1	1,3	1,7	0,3	0	0
A	0	0	2	0,7	0,3	1,7	**2,7**	1,3	1	0,4	1	1,3	1,3	0
U	0	0	0,7	1,7	0,3	1,3	**2,7**	2,3	1	0,7	1,7	2	1	1
U	0	0	0,3	0,3	1,3	1	2,3	**2,3**	2	0,7	1,7	2,7	1,7	1
G	0	0	0	1,3	0	1	1	2	**3,3**	2	1,7	1,3	2,3	2,7
A	0	0	0	0	1	0,3	0,7	0,7	2	3	1,7	1,3	2,3	2
C	0	1	1	0,7	1	2	0,7	1,7	1,7	3	2,7	1,3	1	2
G	0	0	0,7	1	0,3	0,7	1,7	0,3	2,7	1,7	2,7	2,3	1	2
G	0	0	0	1,7	0,7	0,3	0,3	1,3	1,3	2,3	1,3	2,3	2	2

Figura 1.4: Aplicación del algoritmo Smith-Waterman a las secuencias A y B.

En este caso el puntaje de similitud es 3,3 y el par de segmentos con máxima similitud es:

- $G\ C\ C\ A\ U\ U\ G$
- $G\ C\ C\ _\ U\ C\ G$

Capítulo 2

Arquitectura cluster de multicores

En este Capítulo se describe a las arquitecturas paralelas de clusters y su correspondiente evolución a cluster de multicores.

2.1. Clusters

El término cluster se aplica a los conjuntos de computadoras construidos mediante la utilización de componentes de hardware estándar y que se comportan como si fuesen una única computadora [7]. En la actualidad, desempeñan un papel importante en la solución de problemas de las ciencias, las ingenierías y del comercio moderno [8]. La tecnología de clusters ha evolucionado en apoyo de actividades que van desde aplicaciones de supercómputo y software de misión crítica, servidores web y comercio electrónico, hasta bases de datos de alto rendimiento, entre otros usos.

La computación con clusters surgió como resultado de la convergencia de varias tendencias que aún hoy se mantienen, entre las que se incluyen la disponibilidad de redes de alta velocidad y procesadores económicos de alto rendimiento, el desarrollo de herramientas de software para cómputo distribuido de alto rendimiento, así como la creciente demanda de potencia computacional para aplicaciones que la requieran [9].

La construcción de los nodos de un cluster es relativamente fácil y económica debido a su flexibilidad: pueden tener todos la misma configuración de hardware y sistema operativo (cluster homogéneo), o tener diferente hardware y/o sistema operativo (cluster heterogéneo). Esta característica constituye un factor importante en el análisis del rendimiento que se puede obtener de un cluster como máquina paralela [10].

El paradigma de programación paralela natural para los clusters es el modelo de pasaje de mensajes [11]. La primera librería de pasaje de mensajes utilizada a gran escala fue Parallel Virtual Machine (PVM), la cual constituyó una pieza clave en el éxito de las arquitecturas de cluster como máquinas paralelas. Sin embargo, la dificultad para lograr alto rendimiento y la no definición de un estándar terminaron llevándola al desuso. Subsecuentemente, se definió la librería Message-Passing Interface (MPI), la cual se ha convertido en la librería estándar de pasaje de mensajes y, a su vez, en la más utilizada [9].

2.2. Clusters de multicores

En las últimas décadas, los procesadores mejoraron su rendimiento según la ley de Moore, fundamentalmente por dos factores tecnológicos. Primero el aumento de la frecuencia alcanzable por el reloj y en segundo lugar, el creciente número de transistores en un chip, que en conjunto con mejoras en los compiladores, incrementaron el número de instrucciones ejecutables por unidad de tiempo. Sin embargo, esta mejora se vio limitada por la presencia de niveles de generación de calor y consumo de energía que resultaron insostenibles. La solución a este problema consistió en cambiar a un nuevo paradigma de diseño de microprocesadores: el procesador de múltiples núcleos [12, 13].

Un procesador de múltiples núcleos (*multicore*) integra dos o más núcleos computacionales dentro de un único chip. Si bien estos núcleos son más simples y menos veloces, al combinarlos permiten mejorar el rendimiento global del procesador y al mismo tiempo hacerlo más eficiente energéticamente [3, 4].

La incorporación de los procesadores multicore a las arquitecturas de clusters tradicionales ha llevado a éstos a una nueva etapa, dando nacimiento a una arquitectura paralela híbrida conocida como cluster de multicores [14]. En este tipo de arquitecturas, la comunicación entre las diferentes unidades de procesamiento resulta ser heterogénea [15]. Las comunicaciones pueden clasificarse en inter-nodo o intra-nodo. Las comunicaciones inter-nodo se dan entre aquellos núcleos que residen en distintos nodos mediante el envío de mensajes a través de la red de interconexión que une los nodos. Las comunicaciones intra-nodo se producen entre los núcleos que se encuentran dentro del mismo nodo a través de los diferentes niveles de memoria que comparten. De acuerdo a la ubicación de los núcleos, las comunicaciones intra-nodo pueden a su vez clasificarse en tres grupos: caché-compartida (*shared-cache*), intra-socket e inter-socket. Las comunicaciones caché-compartida se realizan entre núcleos que comparten

un mismo nivel de memoria caché (por ejemplo: L2). Las comunicaciones intra-socket se producen entre núcleos que están dentro de un mismo socket pero que no comparten ningún nivel de caché. Por último, las comunicaciones inter-socket se dan entre núcleos que residen en el mismo nodo pero en diferentes sockets. En la Figura 2.1 se muestra un esquema de los diferentes tipos de comunicación que pueden presentarse en un cluster de multicores.

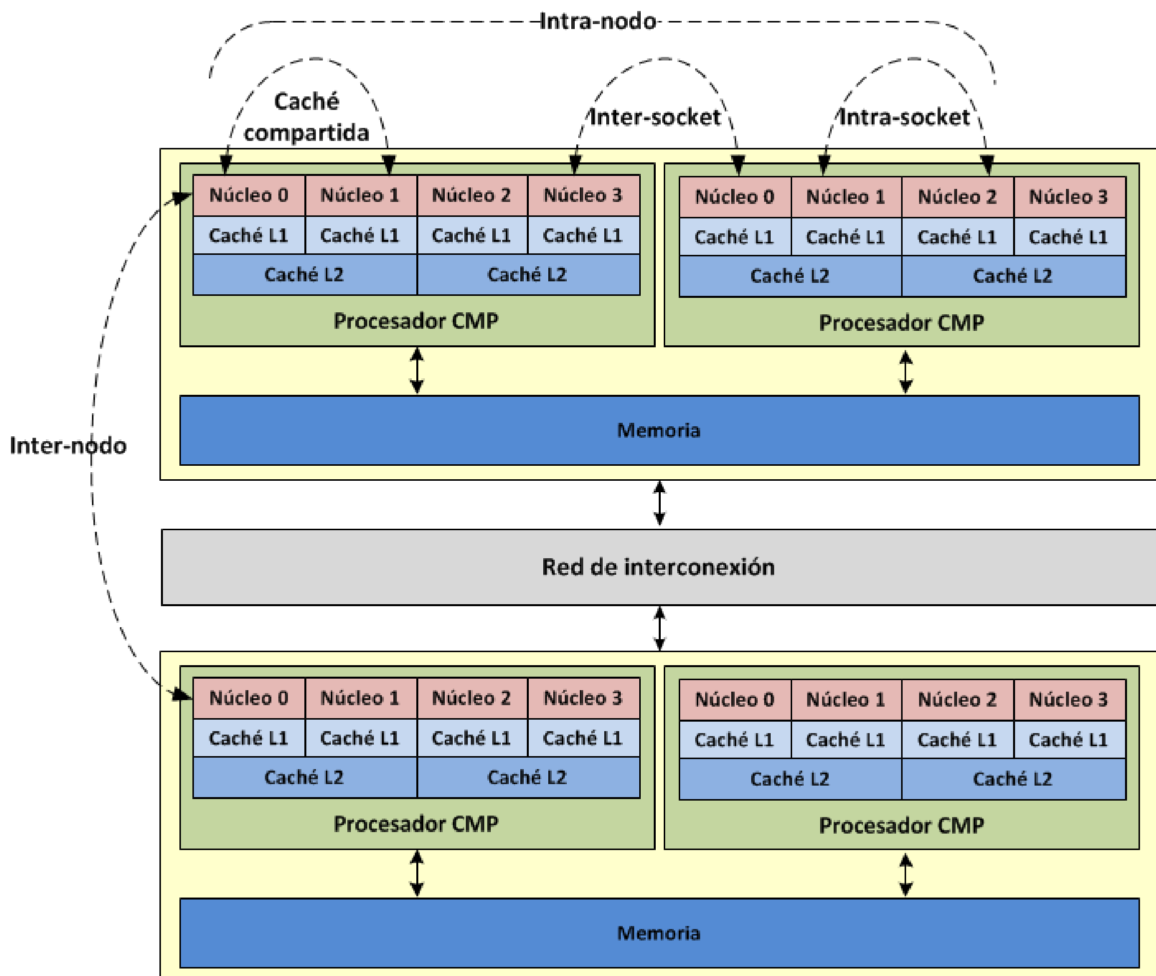

Figura 2.1: Esquema de los diferentes tipos de comunicación que se pueden presentar en una arquitectura de cluster de multicores.

Los clusters de multicores son arquitecturas híbridas, ya que combinan memoria compartida con memoria distribuida. Teniendo en cuenta que el aprovechamiento de la jerarquía de memoria presente en estas arquitecturas incide directamente sobre el rendimiento alcanzable por los mismos, se desarrolló un nuevo paradigma de programación paralela: el paradigma híbrido. Este paradigma combina características del modelo de memoria compartida con el de pasaje de mensajes. En la actualidad, com-

binaciones de la librería MPI junto a las librerías Pthreads u OpenMP son utilizadas para programar según este paradigma. El modelo de pasaje de mensajes, a través del estándar MPI, también puede ser utilizado para la programación de clusters de multicores. En este caso, los procesos que se ejecutan en el mismo nodo simulan memoria distribuida sobre memoria compartida.

La comparación de rendimiento de los paradigmas de programación paralela para clusters de multicores es un tema ampliamente estudiado por la comunidad científica [16, 17, 18, 19]. Si bien los resultados son diversos, en la actualidad la tendencia parece ser hacia el paradigma de pasaje de mensajes a través del uso de MPI. Las razones que motivan esta tendencia son el hecho de que MPI es un estándar definido y ampliamente usado por la comunidad científica junto al desarrollo de optimizaciones en sus implementaciones para trabajar sobre ambientes de memoria compartida [20].

Capítulo 3

Modelo de programación pasaje de mensajes y estándar MPI

En este Capítulo se describe el modelo de programación paralela de pasaje de mensajes y el estándar MPI para programar bajo el mismo.

3.1. Modelo de programación pasaje de mensajes

En el modelo de programación de pasaje de mensajes cada proceso tiene su propio espacio de direcciones y la comunicación y sincronización entre ellos se da a través del envío y la recepción de mensajes.

El modelo de pasaje de mensajes tiene dos grandes ventajas. La primera de ellas es que los programas son altamente portables. Prácticamente cualquier conjunto de computadoras podría ser usado para ejecutar un programa escrito con este modelo. La segunda ventaja consiste en el control explícito que tiene el programador sobre la ubicación de los datos en la memoria. Como el acceso y el manejo de la memoria inciden directamente sobre el desempeño, esta habilidad le permite al programador poder lograr alto rendimiento. La mayor desventaja de este modelo es que el programador no sólo no puede desligarse de detalles como la ubicación de los datos en la memoria y el orden de las sentencias de comunicación sino que está obligado a prestar atención a ellos [11].

3.2. Estándar MPI

Message-Passing Interface (MPI) es una estandarización de una especificación de librería para pasaje de mensajes. MPI define la sintaxis y la semántica de un conjunto de operaciones de comunicación y puede ser utilizada con los lenguajes C, C++, Fortran-77 y Fortran-95. En la actualidad, existen dos versiones del estándar MPI: MPI-1 define operaciones de comunicación estándar y se basa en un modelo estático de procesos. MPI-2 extiende a MPI-1 y provee soporte adicional para manejo dinámico de procesos, comunicación *one-sided* y E/S paralela. Es importante resaltar que MPI es una especificación de una interfaz para la sintaxis y la semántica de un conjunto de operaciones de comunicación y que no da detalles de implementación de las mismas. Por lo tanto, diferentes librerías MPI podrían emplear distintas implementaciones, posiblemente utilizando optimizaciones específicas para cada plataforma de hardware. De todas maneras, como la interfaz es estándar, la portabilidad de los programas está asegurada [13]. Entre las implementaciones MPI más utilizadas se encuentran MPICH [21], LAM/MPI [22] y OpenMPI [23].

Si bien MPI es una interfaz amplia y compleja, con sólo seis funciones se puede resolver una gran cantidad de problemas [11]. En el Cuadro 3.1 se detallan las seis funciones MPI básicas. Más información sobre estas funciones en el Apéndice A.

Función MPI	**Objetivo**
MPI_INIT	Inicia una sesión MPI
MPI_FINALIZE	Finaliza una sesión MPI
MPI_COMM_SIZE	Determina el número de procesos de la sesión MPI actual
MPI_COMM_RANK	Determina el identificador del proceso
MPI_SEND	Envía un mensaje
MPI_RECV	Recibe un mensaje

Cuadro 3.1: Funciones MPI básicas.

3.2.1. Operaciones de comunicación no bloqueantes

Las operaciones de comunicación descritas en la Sección 3.2 son bloqueantes. Esto significa que el control sólo retorna al proceso llamador cuando todos los recursos involucrados pueden ser reusados, como por ejemplo los buffers especificados en la invocación. Los protocolos bloqueantes incurren en cierto overhead para poder garantizar esta corrección semántica. Si en lugar de quedar bloqueado a la espera de que

la operación se complete, el proceso llamador recupera el control inmediatamente, entonces podría realizar cualquier cómputo que no dependa de los datos de la operación de comunicación pendiente. Luego, este proceso puede verificar si la operación de comunicación se completó y, si es necesario, esperar a que lo haga. Este tipo de protocolos se conoce como no bloqueantes y proveen operaciones de envío y recepción rápidas a costo de que el programador verifique la corrección semántica [24].

MPI provee variantes no bloqueantes de las funciones de envío y recepción: MPI_ISEND y MPI_IRECV. Los llamados a estas funciones no esperan a que la comunicación se complete para regresar. Para poder verificar si una operación se completó, MPI cuenta con la función MPI_WAIT. En el Cuadro 3.2 se describen estas funciones. Más información sobre estas funciones en el Apéndice A.

Función MPI	**Objetivo**
MPI_ISEND	Inicia el envío de un mensaje
MPI_IRECV	Inicia la recepción de un mensaje
MPI_WAIT	Espera a que la operación especificada en el objeto Request se complete

Cuadro 3.2: Funciones MPI para comunicaciones no bloqueantes.

3.2.2. Comunicadores

Un comunicador se emplea para definir un conjunto de procesos que se comunican entre ellos. Este conjunto de procesos compone lo que se conoce como *dominio de comunicación*. En general, como todos los procesos podrían necesitar comunicarse entre sí, MPI define un comunicador predeterminado llamado MPI_COMM_WORLD, el cual involucra a todos los procesos del programa. Sin embargo, muchas veces se desea que los procesos sólo se comuniquen por grupos, posiblemente en forma solapada. Una manera de garantizar que los mensajes dentro de un grupo nunca interfieran con los de algún otro es utilizando un comunicador para cada grupo particular [24].

3.2.3. Operaciones de comunicación colectivas

MPI provee un conjunto extensivo de funciones que realizan operaciones de comunicación colectivas comúnmente usadas. Todas estas funciones toman como argumento un comunicador que define el grupo de procesos involucrados en la operación de comunicación. Todos los procesos pertenecientes a este comunicador participan en la operación y es necesario que todos ellos invoquen a la función correspondiente [24].

En el Cuadro 3.3 se describen las funciones MPI más relevantes para comunicaciones colectivas. Más información sobre estas funciones en el Apéndice A.

Función MPI	**Objetivo**
MPI_BARRIER	Sincroniza todos los procesos
MPI_BCAST	Envía datos de un proceso a todos los demás
MPI_GATHER	Agrupa datos de todos los procesos en un único proceso
MPI_SCATTER	Reparte datos de un proceso a todos los demás
MPI_REDUCE	Realiza una operación de reducción todos a uno

Cuadro 3.3: Funciones MPI más relevantes para comunicaciones colectivas.

Capítulo 4

Evaluación de sistemas paralelos

En este Capítulo se describen diferentes factores que producen overhead en los programas paralelos además de un conjunto de métricas tradicionales que permiten evaluar el rendimiento de sistemas paralelos.

4.1. Concepto de sistema paralelo

Un algoritmo secuencial suele ser evaluado a partir de su tiempo de ejecución, el cual se expresa como una función del tamaño de su entrada. Evaluar a un algoritmo paralelo de la misma manera que a uno secuencial sería injusto, ya que el tiempo de ejecución no sólo depende del tamaño de la entrada sino también del número de unidades de procesamiento utilizadas y de las velocidades de las mismas para computar y comunicar datos. Es por ello que un algoritmo paralelo no puede ser evaluado en forma aislada de la arquitectura donde se ejecuta sin perder precisión. Un *sistema paralelo* es la combinación de un algoritmo junto a la arquitectura paralela sobre la cual se ejecuta [24].

4.2. Fuentes de overhead en programas paralelos

Al duplicar los recursos de hardware, se espera razonablemente que un programa se ejecute dos veces más rápido. Sin embargo, esto no suele ocurrir en los programas paralelos debido a que los mismos incurren en diferentes overheads asociados a la administración del paralelismo.

Además del cómputo asociado al problema a resolver, un programa paralelo puede dedicar tiempo a la interacción entre procesos, al ocio, y al exceso de cómputo

(cómputo que no es realizado por la versión secuencial) [24].

4.2.1. Interacción entre procesos

Cualquier implementación paralela requiere que sus unidades de procesamiento interactúen y comuniquen datos (por ejemplo, resultados intermedios). La interacción entre procesos es quizás la fuente más significativa de overhead en el procesamiento paralelo.

4.2.2. Ocio

Las unidades de procesamiento en un sistema paralelo pueden estar ociosas debido a diferentes razones como desbalance de carga, sincronización, y presencia de código serial en un programa.

Si diferentes unidades de procesamiento tienen distintas cargas de trabajo, entonces algunas de ellas pueden estar ociosas mientras otras trabajan sobre el problema. Lo anterior puede ocurrir, por ejemplo, cuando la generación de tareas es dinámica. En dicho caso, se hace difícil estimar el tamaño de las subtareas asignadas a las unidades de procesamiento, lo cual puede causar que las cargas de trabajo no sean uniformes.

En algunos programas paralelos, las unidades de procesamiento deben sincronizar en determinados puntos de la ejecución. Si no todas están listas al mismo tiempo, entonces las que lo estén primero estarán ociosas hasta que el resto lo haga.

Un programa paralelo podría contener partes que deben ser ejecutadas secuencialmente por una unidad de procesamiento. Durante ese cómputo secuencial, las otras unidades de procesamiento deben esperar.

4.2.3. Exceso de cómputo

El algoritmo secuencial más rápido para resolver un problema puede ser difícil o imposible de paralelizar, lo cual obliga a utilizar un algoritmo paralelo basado en un algoritmo secuencial más lento pero que garantice un grado de concurrencia más alto. De esta forma, la versión paralela incurre en un cierto overhead debido al exceso de cómputo que debe realizar en relación a la mejor versión secuencial.

También puede ocurrir que un programa paralelo deba realizar más cómputo que la mejor versión secuencial, ya que al estar distribuido entre diferentes unidades de procesamiento, no resulta posible reutilizar resultados intermedios. Esto implica que

algunos cálculos sean realizados múltiples veces por diferentes unidades de procesamiento.

4.3. Métricas de rendimiento para sistemas paralelos

Es importante estudiar el rendimiento de programas paralelos con el fin de determinar el mejor algoritmo, evaluar plataformas de hardware, y analizar los beneficios del paralelismo. Para ello se han definido un conjunto de métricas.

4.3.1. Tiempo de ejecución

El tiempo de ejecución secuencial de un programa es el tiempo que pasa entre el inicio y el fin de su ejecución en una computadora utilizando una única unidad de procesamiento. El tiempo de ejecución paralelo es el tiempo que transcurre desde el momento en que el procesamiento paralelo comienza hasta que el último elemento de procesamiento finaliza su ejecución. Se expresa el tiempo de ejecución secuencial como T_{s} y el tiempo de ejecución paralelo como T_{p} [24].

4.3.2. Speedup

El *speedup* refleja la ganancia que se obtiene al paralelizar la solución a un problema. Se define como el cociente entre el tiempo requerido por la solución secuencial utilizando una única unidad de procesamiento y el tiempo requerido por la solución paralela de interés utilizando más de una unidad de procesamiento. Se denota al speedup como *S(p)*, siendo p el número de unidades de procesamiento utilizadas. La Ecuación 4.1 indica cómo calcular esta métrica.

$$S(p) = \frac{T_s}{T_p} \tag{4.1}$$

En otras palabras, el speedup indica cuántas veces más rápido se obtiene la solución al problema utilizando la solución paralela con p unidades de procesamiento con respecto a la solución secuencial utilizando una única unidad de procesamiento. Se asume que las p unidades de procesamiento empleadas por la solución paralela son idénticas a la utilizada por la versión secuencial. Cuando se hace referencia a la solución secuencial se debe tener en cuenta que puede existir más de una que resuelva el problema. Se debe elegir el algoritmo secuencial que soluciona el problema en la

menor cantidad de tiempo. De otra manera la comparación con el algoritmo paralelo no sería justa [9, 24].

4.3.2.1. ¿Cuál es el máximo speedup alcanzable?

En teoría, el máximo speedup alcanzable con p unidades de procesamiento es p, lo que se conoce como *speedup lineal*. Para ello cada unidad de procesamiento debería tardar $\frac{T_s}{p}$ unidades de tiempo, siendo T_s el tiempo requerido por la mejor solución secuencial para resolver el problema. Para obtener un speedup superior a p, es necesario que cada unidad de procesamiento tarde menos de $\frac{T_s}{p}$ unidades de tiempo. En ese caso, una única unidad de procesamiento podría emular las p unidades de procesamiento y resolver el problema en menos de T_s unidades de tiempo. Esto representa una contradicción, dado que el speedup, por definición, es calculado a partir de la mejor solución secuencial. Si T_s es el tiempo requerido por la solución secuencial más rápida, entonces el problema no puede ser resuelto en menos de T_s empleando una única unidad de procesamiento .

Sin embargo, en la práctica, a veces se observan speedups superiores a p, lo que se conoce como *speedup superlineal*. Esto se puede deber a que el trabajo que debe realizar la solución secuencial es mayor que el correspondiente a la solución paralela o también debido a características de hardware que ponen en desventaja a la solución secuencial [9, 24].

4.3.3. Eficiencia

La eficiencia es una medida que refleja cuan bien utilizadas son las unidades de procesamiento para resolver el problema. Se define como el cociente entre el speedup y p, siendo éste el número de unidades de procesamiento utilizados, y se denota mediante la letra E, como se indica en la Ecuación 4.2.

$$E = \frac{S(p)}{p} \tag{4.2}$$

En la teoría, se puede obtener una eficiencia igual a 1 si el speedup obtenido con p unidades de procesamiento es igual a p. En la práctica, debido a los costos que implican la sincronización y la comunicación de las tareas, el speedup suele ser menor a p. Esto lleva a la eficiencia a ser menor a 1 [9].

4.3.4. Escalabilidad

La *escalabilidad* de un sistema paralelo es una medida de su habilidad para utilizar un número creciente de unidades de procesamiento en forma eficiente.

Diversos métodos han sido propuestos para poder medir la escalabilidad de los sistemas paralelos. Entre ellos se encuentran los modelos de isoeficiencia, speedup escalado y fracción serie f. Probablemente el más utilizado debido a su efectividad sea el primero de ellos. El modelo de isoeficiencia establece que un sistema es escalable si éste es capaz de mantener su eficiencia en un valor fijo al incrementar tanto el número de unidades de procesamiento como el tamaño del problema. La *función de isoeficiencia* determina la tasa a la cual debe crecer el tamaño del problema, si se utilizan más unidades de procesamiento, para evitar una caída de la eficiencia del sistema [24].

El modelo de isoeficiencia se basa en las siguientes dos consideraciones:

- Para un tamaño de problema fijo, existe un límite en el incremento del speedup por el uso de un número creciente de unidades de procesamiento. Esto se debe a que, a medida que se utilizan más unidades de procesamiento, aumenta el overhead presente en la solución paralela, lo que provoca una caída de la eficiencia del sistema.

- La eficiencia de la mayoría de los sistemas paralelos aumenta si se incrementa el tamaño de problema manteniendo fijo el número de unidades de procesamiento empleadas. El aumento en la eficiencia se debe a que la proporción de overhead crece en menor medida a la proporción de cómputo al incrementar el tamaño del problema.

Capítulo 5

Algoritmos para el alineamiento de secuencias de ADN por el método Smith-Waterman

En este Capítulo se presentan diferentes algoritmos para el alineamiento de secuencias de ADN por el método Smith-Waterman con el objetivo de determinar el puntaje de similitud entre dos secuencias de ADN. Esto significa que no se tiene en cuenta el proceso de retroceso para obtener el segmento que representa el alineamiento óptimo (no se realiza el punto 4 del algoritmo explicado en la Sección 1.4).

5.1. Algoritmo secuencial

Existe una dependencia de datos para calcular los valores de la matriz, la cual se muestra en la Figura 5.1. Para obtener $H_{i,j}$ se requiere el resultado de $H_{i-1,j-1}$ (H_d en la Figura 5.1) y también se necesita saber el puntaje al considerar un gap en la fila i y otro en la columna j. Esta restricción lleva a calcular los valores de H de arriba hacia abajo y de izquierda a derecha ($H_{1,1}$, $H_{1,2}$, $H_{1,3}$, $\ldots$, $H_{2,1}$, $H_{2,2}$, $H_{2,3}$, ...).

Teniendo en cuenta que no se realiza el paso 4 del algoritmo, no es necesario almacenar la matriz H completa. En su lugar se necesita:

- Un vector h de longitud $M+1$ que mantiene en cada posición el valor obtenido en la última fila procesada sobre esa columna. En la Ecuación 5.1 se indican los valores de h en el ejemplo de la Figura 5.1.

$$h_k = \begin{cases} H_{i,k} & k < j-1 \\ H_{i-1,k} & k \geq j-1 \end{cases} \tag{5.1}$$

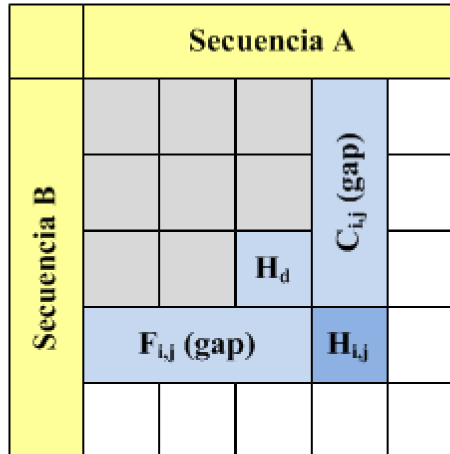

Figura 5.1: Esquema de dependencias de datos

- Un elemento e para guardar en forma temporal el último valor calculado en la fila que se está procesando. En la Figura 5.1, $e = H_{i,j-1}$.

- Un vector c de longitud *M+1* que mantiene en cada posición el máximo puntaje considerando un gap en esa columna. En la Ecuación 5.2 se indican los valores de c en el ejemplo de la Figura 5.1.

$$c_k = \begin{cases} C_{i,k} & k < j \\ C_{i-1,k} & k \geq j \end{cases} \tag{5.2}$$

- Un elemento f que mantiene el máximo puntaje considerando un gap en la fila que se está procesando. En el ejemplo de la Figura 5.1, $f = F_{i,j-1}$.

5.2. Algoritmo secuencial por bloques

Una de las técnicas más efectivas y utilizadas para mejorar el rendimiento de un algoritmo consiste en maximizar la localidad temporal y la localidad espacial de los datos. Se dice que un programa tiene localidad espacial de los datos, si el mismo accede a datos en la memoria que se encuentran en posiciones vecinas en puntos sucesivos en el tiempo durante la ejecución. El beneficio de aprovechar la localidad espacial de los datos se da a partir del hecho de que, una vez cargado un bloque de datos en la caché, varios de los siguientes accesos a memoria podrán ser resueltos a partir de éste, evitando de esta manera la ocurrencia de fallos de caché que resulten costosos. Se dice que un programa tiene localidad temporal de los datos, si el mismo accede múltiples veces a datos que se encuentran en las mismas posiciones de memoria en

puntos sucesivos en el tiempo durante la ejecución. Este tipo de programas obtiene una ganancia en su rendimiento ya que ocurre con frecuencia que, una vez cargado un bloque de datos en la caché, las palabras que lo componen serán accedidas múltiples veces antes de que el bloque de datos sea reemplazado [13, 24].

La solución secuencial presentada en la Sección 5.1 calcula los valores de la matriz H, procesando la misma por filas de arriba hacia abajo y, luego, de izquierda a derecha. El procesamiento principal se da sobre el vector h, el cual, como se explicó, mantiene en cada posición el valor de la última fila procesada sobre cada columna. Es claro que ésta solución exhibe localidad espacial de los datos. Sin embargo, no se puede decir lo mismo respecto a la localidad temporal. Si la secuencia A es muy grande, entonces los diferentes bloques de datos en memoria que componen al vector h, estarían entrando y saliendo de la caché en forma continua, perjudicando de ésta manera el rendimiento de la solución. Una forma de evitar esta problemática y, al mismo tiempo, mejorar la localidad temporal de los datos, consiste en procesar la matriz H por bloques verticales, calculando los valores de cada uno por filas de arriba hacia abajo y de cada fila de izquierda a derecha. En la Figura 5.2 se muestra el esquema de resolución empleado en esta solución.

El uso de esta técnica obliga a cambiar algunas de las estructuras de datos empleadas en la solución de la Sección 5.1 y a agregar otra, como se detalla a continuación:

- No es necesario que el vector h posea *M+1* posiciones, sino que resultan suficientes *TBA+1*, siendo *TBA* la cantidad de elementos de la secuencia A que determinan el ancho de los bloques de datos verticales mediante los cuales se procesa la matriz *H*.

- Un único elemento ya no es suficiente para mantener el máximo puntaje considerando un gap en cada fila. Es por ello que el elemento f se reemplaza por un vector f de longitud *N+1*. En la Ecuación 5.3 se indican los valores de f en el ejemplo de la Figura 5.1.

$$f_k = \begin{cases} F_{i,k} & k < j \\ F_{i,k-1} & k \geq j \end{cases} \tag{5.3}$$

- Un vector u de longitud *N+1* que mantiene en cada posición el valor obtenido en la última columna procesada sobre esa fila. En la Ecuación 5.4 se indican los valores de u en el ejemplo de la Figura 5.1.

$$u_k = \begin{cases} H_{i,k} & k < j \\ H_{i,k-1} & k \geq j \end{cases} \tag{5.4}$$

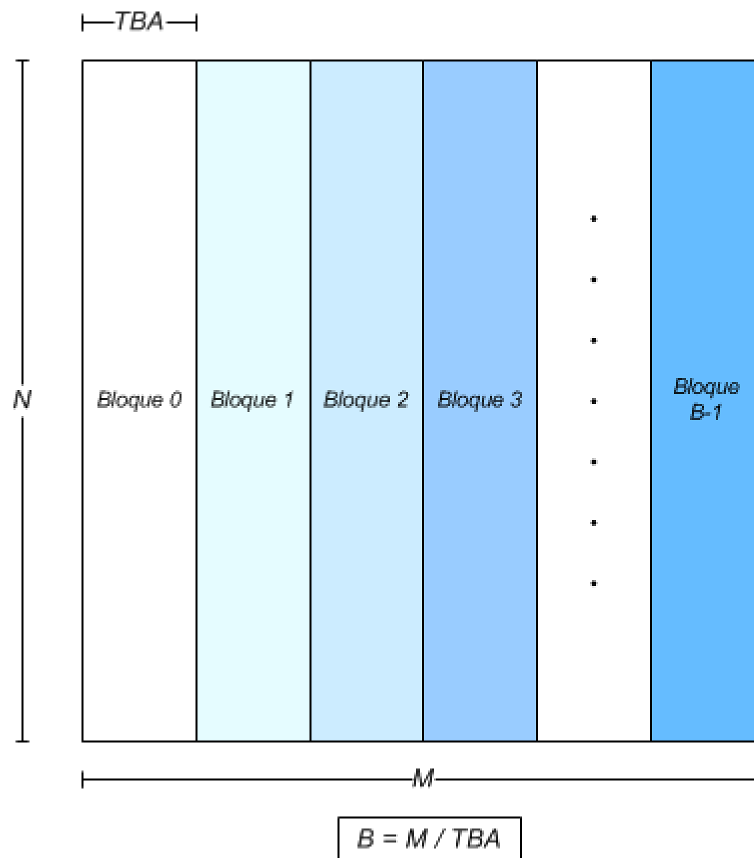

Figura 5.2: Esquema de resolución empleado por la solución secuencial por bloques.

Resulta importante para el rendimiento final de esta solución determinar el valor de TBA. Se debe tener en cuenta que:

- Si el tamaño TBA es muy grande, el tráfico de memoria principal a la caché aumentará, perjudicando el rendimiento de la solución. Desde este punto de vista, TBA debe tender a 1.

- Un valor muy pequeño de TBA, no permitirá maximizar la localidad temporal de los datos, desaprovechando la posibilidad de optimizar el cómputo. Desde este punto de vista, TBA debe tender a M.

Resulta importante encontrar un tamaño de bloque TBA que maximize la localidad temporal de los datos. El tamaño de los distintos niveles de memoria caché es un factor fundamental que se debe tener en cuenta.

5.3. Algoritmo paralelo

La dependencia de datos mencionada en la Sección 5.1 lleva a resolver el problema con un esquema de pipeline en el que P procesos realizan el mismo trabajo sobre diferentes subconjuntos de nucleótidos consecutivos de la primera secuencia (A en Figura 5.1). En cada ciclo el proceso p_i (para $i \in [1, P-1]$) recibe resultados parciales de p_{i-1} y a partir de ellos resuelve parte de su trabajo. El cómputo a realizar en cada ciclo consiste en calcular los valores de un bloque de $(\frac{M}{P}) \times TBB$ elementos de la matriz H, siendo TBB la cantidad de elementos de la secuencia B que determinan la altura de los bloques de datos. Los distintos procesos computan cada bloque de datos mediante sub-bloques verticales de $TBA \times TBB$ elementos. Una vez procesado el bloque de datos correspondiente, el proceso p_i envía resultados parciales a p_{i+1} (excepto el último que no necesita enviarle los resultados a ningún otro). El primer proceso (p_0) sólo realiza su trabajo enviando los resultados parciales (correspondientes a un bloque) a su sucesor. Este proceso también es el responsable de distribuir la primera secuencia (A en la Figura 5.1) en partes iguales entre todos los procesos y enviarle la segunda secuencia (B en la Figura 5.1) completa a todos ellos. En la Figura 5.3 se muestra el esquema de resolución empleado en esta solución.

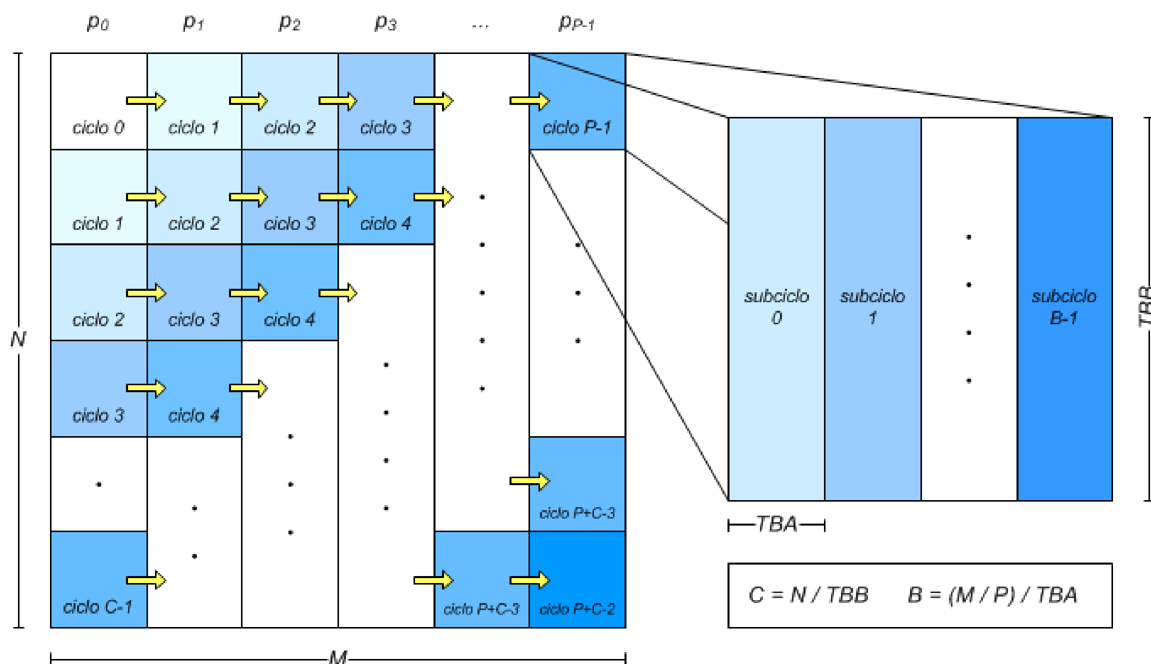

Figura 5.3: Esquema de resolución empleado por la solución paralela.

Un punto importante de esta solución es seleccionar el valor del tamaño TBB, teniendo en cuenta que:

- Recién se comienza a aprovechar al máximo el paralelismo del pipeline cuando han pasado *P-1* ciclos, es decir, cuando a todas los procesos les ha llegado trabajo. Cuanto más grande es *TBB*, mayor es el tiempo que tarda en llenarse el pipeline y, por consiguiente, menor el aprovechamiento del mismo. Desde este punto de vista, *TBB* debe tender a 1.

- Si el tamaño *TBB* es muy pequeño, los procesos pasan más tiempo comunicando resultados parciales que procesando información. Desde este punto de vista, *TBB* debe tender a *N*.

Se debe encontrar un tamaño *TBB* que permita solapar la comunicación de los datos con el procesamiento de los mismos. El tamaño óptimo no sólo depende de las características de la arquitectura utilizada, sino también del modelo de comunicación empleado y el tamaño del problema a resolver.

5.3.1. Modelo para el cálculo del tamaño *TBB* óptimo

En esta Sección se define una función que permite calcular el tamaño de bloque *TBB* que resulta óptimo para el algoritmo paralelo presentado en la Sección 5.3, asumiendo que todas las unidades de procesamiento de la arquitectura de soporte poseen la misma potencia de cómputo.

Se asumen los siguientes valores constantes de la arquitectura de soporte:

- t_s: tiempo de startup promedio para enviar un mensaje,

- t_b: tiempo promedio requerido para enviar un elemento del bloque de datos a comunicar,

- t_c: tiempo requerido para calcular el valor de un elemento de la matriz.

También se asumen los siguientes parámetros de entrada, relacionados con el tamaño de la arquitectura empleada y del problema a resolver:

- N es la longitud de las secuencias A y B,

- P es la cantidad de núcleos empleados.

Para determinar el valor óptimo de *TBB*, primero se debe encontrar una función T que represente el tiempo de ejecución del algoritmo paralelo desde el momento en que el primer proceso del pipeline comienza a trabajar hasta que el último termina. Ante

todo, se define a *CB* como la cantidad de bloques en que se divide la secuencia B, y se determina por medio de la Ecuación 5.5.

$$CB = \frac{N}{TBB} \tag{5.5}$$

Desde el punto de vista del último proceso del pipeline, la función T está dada por el tiempo de procesar cada uno de los *CB* bloques (*TPB*) y el de sus respectivas comunicaciones (*TCB*), más el tiempo en que tarda en llenarse el pipeline, lo que equivale al tiempo de procesar *P-1* bloques y sus correspodientes comunicaciones. Esta función se define en la Ecuación 5.6.

$$T = (P - 1 + CB) \times (TPB + TCB) \tag{5.6}$$

El tiempo requerido para procesar un bloque de tamaño *TBB* es aproximadamente igual para todos los núcleos. Este valor está dado por la Ecuación 5.7.

$$TPB = \frac{N}{P} \times TBB \times t_c \tag{5.7}$$

Para resolver cada bloque de datos, los procesos reciben de su antecesor en el pipeline (a excepción del primero) un vector de tamaño $2 \times TBB$ que contiene resultados parciales. De esta forma, el tiempo requerido para las comunicaciones relacionadas con cada bloque está dado por la Ecuación 5.8.

$$TCB = t_s + (2 \times TBB \times t_b) \tag{5.8}$$

Al reemplazar en la Ecuación 5.6 por lo definido en las Ecuaciones 5.5, 5.7 y 5.8, se obtiene la Ecuación 5.9.

$$T = (P - 1 + \frac{N}{TBB}) \times ((\frac{N}{P} \times TBB \times t_c) + (t_s + 2 \times TBB \times t_b)) \tag{5.9}$$

El valor óptimo de *TBB* es aquel que minimice el valor de la función T. Para esto se debe derivar la función T respecto a *TBB*, como se muestra en la Ecuación 5.10, e igualar esta última a 0. El valor de *TBB* que cumpla esta igualdad será el que resulte óptimo.

$$T' = N \times t_c + 2 \times P \times t_b - \frac{N}{P} \times t_c - 2 \times t_b - N \times t_s \times TBB^{-2} \tag{5.10}$$

En la Ecuación 5.11 se muestra la función empleada para determinar el tamaño

de bloque *TBB* óptimo de acuerdo a las características de la arquitectura, el modelo de comunicación y el tamaño del problema a resolver.

$$TBB(t_s, t_b, t_c, N, P) = \sqrt{\frac{N \times t_s}{(N \times t_c \times (1 - P^{-1}) + (2 \times t_b \times (P - 1)))}} \tag{5.11}$$

Capítulo 6

Trabajo experimental

En este Capítulo se describen las características de la arquitectura de soporte utilizada junto al trabajo experimental realizado.

6.1. Arquitectura de soporte

Para realizar la experimentación se ha utilizado un cluster de multicores Blade de 8 hojas [25, 26]. Cada una de ellas cuenta con 2 procesadores quad core Intel Xeón e5405 de 2.0 GHz, 10 Gb de memoria RAM y caché L2 de 2 x 6Mb entre par de núcleos. El sistema operativo es GNU/Linux Fedora 12 de 64 bits.

6.2. Algoritmos utilizados

Las pruebas se realizaron utilizando tres algoritmos:

- el algoritmo secuencial descrito en la Sección 5.1.
- el algoritmo secuencial por bloques descrito en la Sección 5.2.
- el algoritmo paralelo descrito en la Sección 5.3, siendo P es el número de núcleos usados.

Los algoritmos fueron desarrollados utilizando el lenguaje C (compilador gcc versión 4.4.2) con la librería OpenMPI (compilador mpicc versión 1.4.3) para el pasaje de mensajes.

6.3. Pruebas realizadas

Las pruebas realizadas fueron divididas en dos fases. El objetivo de la primera fase fue la ejecución de las soluciones secuenciales mientras que el de la segunda fue la ejecución de la solución paralela.

6.3.1. Primera fase de prueba

Las soluciones secuenciales fueron probadas utilizando diferentes tamaños de problema: $N = \{65536, 131072, 262144, 524288, 1048576, 2097152\}$. En el caso de la solución secuencial por bloques, se utilizaron diferentes tamaños de bloque TBA para poder determinar de forma empírica el valor que resulta óptimo: $TBA = \{1024, 2048, 4096, 8192, 16384, 32768\}$. Estos valores fueron seleccionados teniendo en cuenta características físicas de la arquitectura de soporte. Cada instancia de prueba particular fue ejecutada diez veces calculando el tiempo de ejecución promedio para el análisis de los resultados.

6.3.2. Segunda fase de prueba

Antes de realizar las pruebas correspondientes a ésta fase, se determinaron las diferentes cantidades de núcleos y de tamaños de problema a utilizar en las mismas. A partir de estas, se calculó el tamaño de bloque TBB que resulta óptimo de acuerdo al modelo presentado en la Subsección 5.3.1. En el Cuadro 6.1 se muestra el resultado de éste cálculo (los valores obtenidos fueron aproximados al entero más cercano). En el Apéndice B se describe el procedimiento llevado a cabo para estimar el valor de los parámetros del modelo.

P	***N***					
	65536	131072	262144	524288	1048576	2097152
16	44	44	44	44	44	44
32	43	43	44	44	44	44
64	43	43	43	43	43	43

Table 6.1: Tamaño de bloque TBB óptimo según modelo para diferente cantidad de núcleos y de tamaños de problema.

Para poder validar la efectividad del modelo desarrollado, se completó el diseño del conjunto de pruebas a realizar incorporando la variación del tamaño de bloque TBB utilizado. Con respecto al tamaño de bloque TBA, se utilizó el valor 16384 en

todos los casos que lo permitió la configuración del número de núcleos utilizados y el tamaño de problema seleccionado (éste valor fue identificado empíricamente como óptimo como se describe en la Sección 7.1). En el Cuadro 6.2 se muestra un esquema de las pruebas realizadas en esta fase.

N	***P***					
	16		**32**		**64**	
65536	4096	1	2048	1	1024	1
		16		16		16
		44		43		43
		128		128		128
		512		512		512
131072	8192	1	4096	1	2048	1
		16		16		16
		44		43		43
		128		128		128
		512		512		512
262144	16384	1	8192	1	4096	1
		16		16		16
		44		44		43
		128		128		128
		512		512		512
524288	16384	1	16384	1	8192	1
		16		16		16
		44		44		43
		128		128		128
		512		512		512
1048576	16384	1	16384	1	16384	1
		16		16		16
		44		44		43
		128		128		128
		512		512		512
2097512	16384	1	16384	1	16384	1
		16		16		16
		44		44		43
		128		128		128
		512		512		512
	TBA	***TBB***	***TBA***	***TBB***	***TBA***	***TBB***

Table 6.2: Esquema de pruebas de la segunda fase.

Cada instancia de prueba particular fue ejecutada diez veces calculando el tiempo de ejecución promedio para el análisis de los resultados.

Capítulo 7

Resultados

Una vez finalizadas las fases de pruebas, se calcularon diferentes métricas de rendimiento y se procedió a analizar los resultados obtenidos.

7.1. Resultados de la primera fase de pruebas

Para evaluar el rendimiento de los algoritmos secuenciales desarrollados al escalar el tamaño del problema, se analiza su tiempo de ejecución. En el Cuadro 7.1 se muestran los promedios de los tiempos de ejecución (en segundos) obtenidos con el algoritmo secuencial para los diferentes tamaños de problema (N) utilizados.

N					
65536	**131072**	**262144**	**524288**	**1048576**	**2097512**
131,42	525,09	2099,47	8398,96	33743,03	135026,69

Cuadro 7.1: Promedios de los tiempos de ejecución obtenidos con el algoritmo secuencial para los diferentes tamaños de problema (N) utilizados.

En el Cuadro 7.2 se muestran los promedios de los tiempos de ejecución (en segundos) obtenidos con el algoritmo secuencial por bloques para los diferentes tamaños de problema (N) y de bloque TBA utilizados. Se resalta con fondo amarillo el menor valor obtenido para cada tamaño de problema.

En este cuadro se puede observar que, sin importar el tamaño de problema empleado, el tiempo de ejecución se reduce a medida que se incrementa el tamaño de bloque TBA usado. Esta mejora se mantiene hasta el tamaño 16384, ya que el tiempo de ejecución aumenta considerablemente cuando $TBA = 32768$. Es claro que el valor 16384 le permite a este algoritmo maximizar la localidad temporal y espacial de los datos, con lo cual se lo identifica como el óptimo del tamaño de bloque TBA.

TBA	***N***					
	65536	**131072**	**262144**	**524288**	**1048576**	**2097512**
1024	126,09	504,25	2015,39	8061,89	32234,12	129012,16
2048	125,41	501,42	2004,15	8016,67	32053,27	128291,69
4096	125,01	499,793	1997,64	7989,89	31947,49	127844,410
8192	124,60	498,19	1991,08	7963,22	31841,38	127407,35
16384	124,34	497,06	1986,57	7944,93	31769,27	127118,51
32768	132,09	527,99	2110,97	8440,28	33755,94	135040,97

Cuadro 7.2: Promedios de los tiempos de ejecución (en segundos) obtenidos con el algoritmo secuencial por bloques para los diferentes tamaños de problema (N) y de bloque TBA utilizados.

En el Cuadro 7.3 se comparan los promedios de los tiempos de ejecución (en segundos) obtenidos por los dos algoritmos secuenciales para los diferentes tamaños de problema (N) usados. En el caso del algoritmo secuencial por bloques, se emplearon los valores obtenidos utilizando el tamaño de bloque $TBA = 16384$, ya que éste fue identificado como el óptimo.

Algoritmo	***N***					
	65536	**131072**	**262144**	**524288**	**1048576**	**2097512**
Secuencial	131,42	525,09	2099,47	8398,96	33743,03	135026,69
Secuencial por bloques	124,34	497,06	1986,57	7944,93	31769,27	127118,51

Cuadro 7.3: Comparación de los promedios de los tiempos de ejecución obtenidos por los dos algoritmos secuenciales para los diferentes tamaños de problema (N) usados.

En este cuadro se puede ver claramente que el algoritmo secuencial por bloques obtiene menores tiempos de ejecución, reflejando de esta manera el beneficio de maximizar la localidad temporal y espacial de los datos. La mejora introducida por el algoritmo secuencial por bloques se puede apreciar de mejor manera en la Figura 7.1, la cual muestra el porcentaje de diferencia relativa (pdr) entre ambos algoritmos, calculado de acuerdo a la Ecuación 7.1, para los diferentes tamaños de problema (N) utilizados.

$$pdr = \frac{T(alg._secuencial) - T(alg._secuencial_por_bloques)}{T(alg._secuencial)} \times 100 \tag{7.1}$$

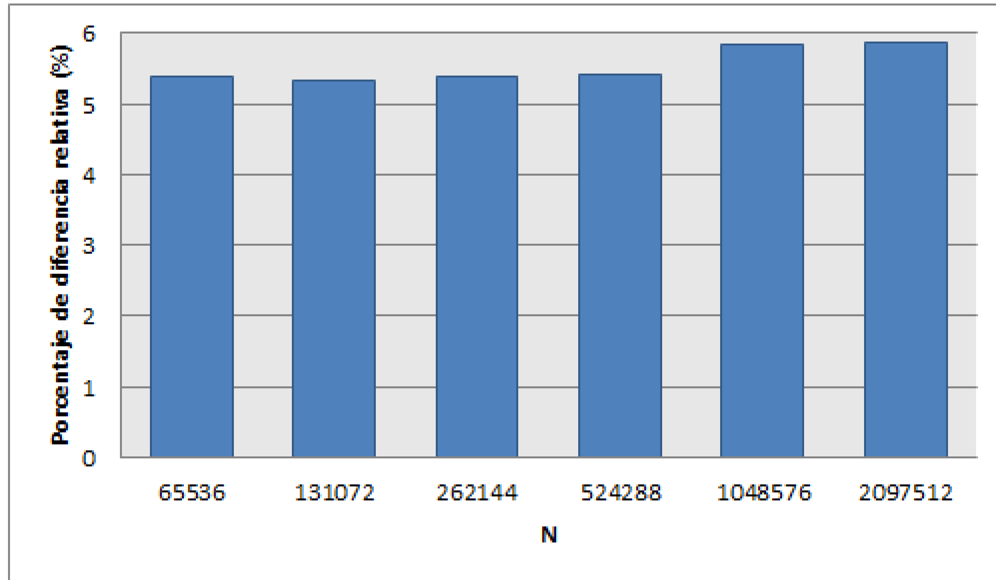

Figura 7.1: Porcentaje de diferencia relativa (pdr) entre los dos algoritmos secuenciales para los diferentes tamaños de problema (N) utilizados.

En el gráfico anterior se puede observar que el empleo de la técnica de procesamiento por bloques permite reducir aproximadamente un 5 % el tiempo de ejecución del algoritmo secuencial original, y que este porcentaje se incrementa levemente a medida que aumenta el tamaño del problema.

7.2. Resultados de la segunda fase de pruebas

Para evaluar el rendimiento del algoritmo paralelo desarrollado, al escalar el tamaño del problema y/o la arquitectura, se analizan las métricas speedup y eficiencia descritas en la Sección 4.3.

En los Cuadros 7.4, 7.5 y 7.6 se muestran los valores de los promedios de los tiempos de ejecución, de speedup y de eficiencia, respectivamente, obtenidos en las pruebas realizadas en la segunda fase. En el Cuadro 7.6 se resalta con fondo amarillo la mejor eficiencia obtenida para cada par de número de unidades de procesamiento (P) y tamaño de problema (N).

P	*TBB*	***N***					
		65536	**131072**	**262144**	**524288**	**1048576**	**2097512**
16	**1**	10,52	38,84	151,78	593,51	2318,10	9221,23
	16	10,38	38,39	151,67	586,13	2307,14	9153,92
	43	10,03	38,34	149,76	585,07	2289,33	9117,87
	128	10,25	39,06	160,55	595,26	2312,98	9122,66
	512	18,23	50,49	168,91	623,18	2403,03	9391,97
32	**1**	7,69	21,08	82,25	307,89	1196,53	4637,97
	16	6,07	20,75	80,48	306,90	1190,13	4645,21
	43/44	5,89	20,43	78,15	302,42	1181,44	4642,54
	128	6,21	21,45	79,67	307,59	1187,93	4640,91
	512	19,06	36,38	101,65	338,72	1255,50	4837,74
64	**1**	6,96	12,36	43,03	161,67	600,27	2349,07
	16	3,61	11,56	41,43	158,83	604,81	2371,57
	44	3,55	11,48	41,09	157,73	617,30	2388,48
	128	3,81	11,90	42,92	158,45	609,69	2363,53
	512	28,56	37,88	73,33	201,33	679,11	2514,64

Cuadro 7.4: Promedios de los tiempos de ejecución (en segundos) obtenidos con el algoritmo paralelo en las pruebas realizadas en la segunda fase.

P	*TBB*	***N***					
		65536	**131072**	**262144**	**524288**	**1048576**	**2097512**
16	**1**	11,82	12,80	13,09	13,39	13,70	13,79
	16	11,98	12,95	13,10	13,55	13,77	13,89
	43	12,40	12,96	13,27	13,58	13,88	13,94
	128	12,13	12,73	12,37	13,35	13,74	13,93
	512	6,82	9,85	11,76	12,75	13,22	13,53
32	**1**	16,17	23,58	24,15	25,80	26,55	27,41
	16	20,48	23,95	24,69	25,89	26,69	27,37
	43/44	21,10	24,33	25,42	26,27	26,89	27,38
	128	20,03	23,17	24,93	25,83	26,74	27,39
	512	6,52	13,66	19,54	23,46	25,30	26,28
64	**1**	17,87	40,21	46,17	49,14	52,93	54,11
	16	34,48	42,99	47,95	50,02	52,53	53,60
	44	35,04	43,28	48,34	50,37	51,46	53,22
	128	32,64	41,77	46,29	50,14	52,11	53,78
	512	4,35	13,12	27,09	39,46	46,78	50,55

Cuadro 7.5: Speedups obtenidos con el algoritmo paralelo en las pruebas realizadas en la segunda fase.

P	*TBB*	*N*					
		65536	**131072**	**262144**	**524288**	**1048576**	**2097512**
16	**1**	0,74	0,80	0,82	0,84	0,86	0,86
	16	0,75	0,81	0,82	0,85	0,86	0,87
	43	0,77	0,81	0,83	0,85	0,87	0,87
	128	0,76	0,80	0,77	0,83	0,86	0,87
	512	0,43	0,62	0,74	0,80	0,83	0,85
32	**1**	0,51	0,74	0,75	0,81	0,83	0,86
	16	0,64	0,75	0,77	0,81	0,83	0,86
	43/44	0,66	0,76	0,79	0,82	0,84	0,86
	128	0,63	0,72	0,78	0,81	0,84	0,86
	512	0,20	0,43	0,61	0,73	0,79	0,82
64	**1**	0,28	0,63	0,72	0,77	0,83	0,85
	16	0,54	0,67	0,75	0,78	0,82	0,84
	44	0,55	0,68	0,76	0,79	0,80	0,83
	128	0,51	0,65	0,72	0,78	0,81	0,84
	512	0,07	0,21	0,42	0,62	0,73	0,79

Cuadro 7.6: Eficiencias obtenidas con el algoritmo paralelo en las pruebas realizadas en la segunda fase.

Para validar la efectividad del modelo desarrollado para calcular el tamaño de bloque *TBB*, se analizan las eficiencias obtenidas en las pruebas realizadas. Las Figuras 7.2, 7.3 y 7.4 componen una representación gráfica del Cuadro 7.6.

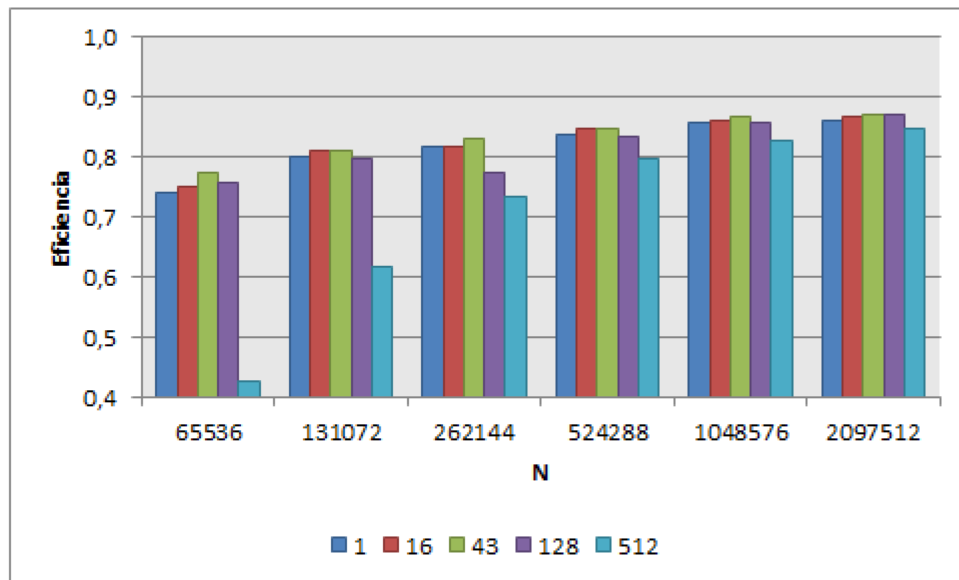

Figura 7.2: Eficiencias obtenidas por el algoritmo paralelo para diferentes tamaños de problema y de bloque al utilizar 16 núcleos de la arquitectura de soporte.

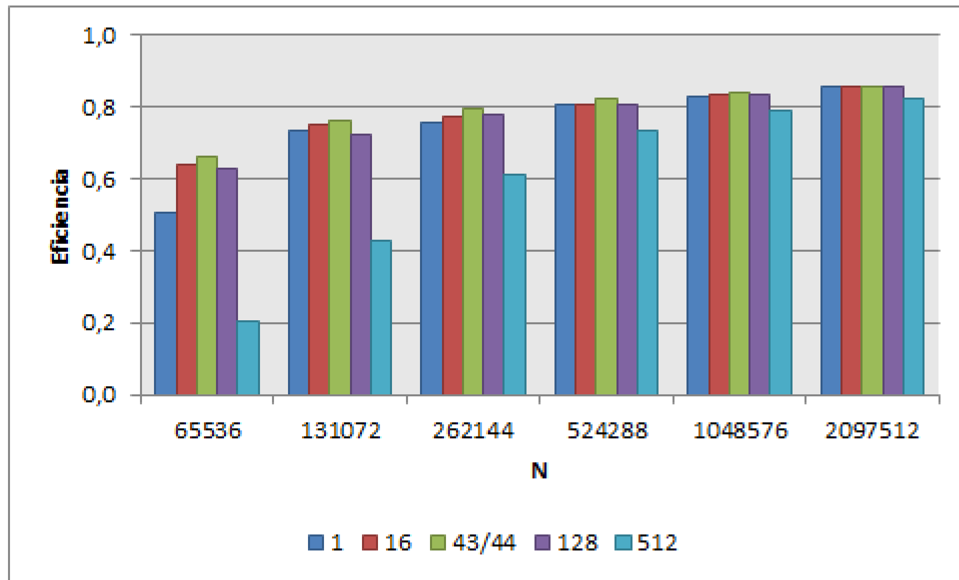

Figura 7.3: Eficiencias obtenidas por el algoritmo paralelo para diferentes tamaños de problema y de bloque al utilizar 32 núcleos de la arquitectura de soporte.

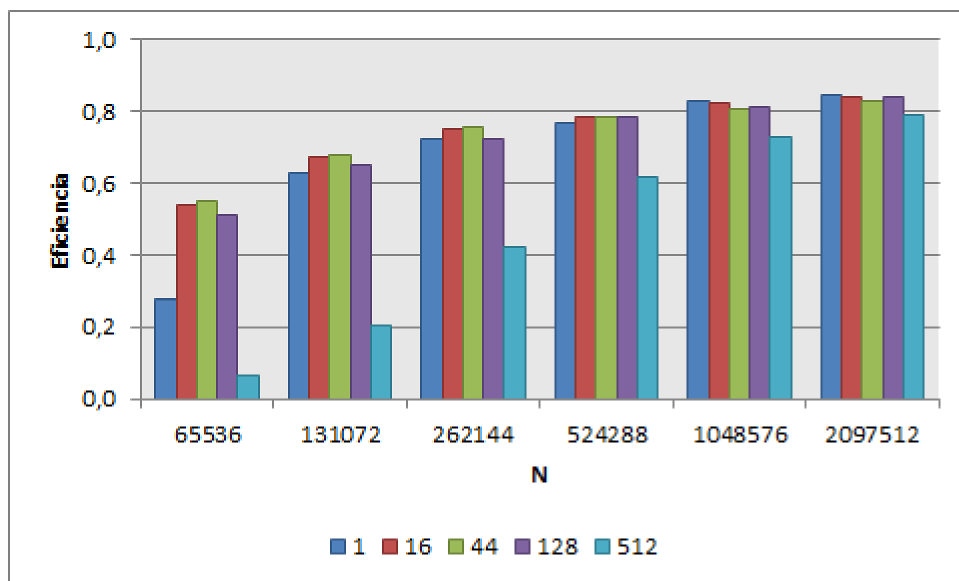

Figura 7.4: Eficiencias obtenidas por el algoritmo paralelo para diferentes tamaños de problema y de bloque al utilizar 64 núcleos de la arquitectura de soporte.

Se puede observar que el algoritmo tiene un rendimiento similar en los tres gráficos y que, en 15 de los 18 casos de pares (P,N), la eficiencia aumenta a medida que se incrementa el tamaño de bloque TBB, alcanzando el valor máximo cuando se emplea el estimado como óptimo en la Subsección 6.3.2. A partir de dicho valor, el incremento del tamaño de bloque utilizado reduce la eficiencia lograda. Esto significa que el mejor

rendimiento se obtiene con los valores calculados a partir del modelo desarrollado. En los tres casos restantes, la mejor eficiencia se obtiene con un tamaño de bloque *TBB* diferente. Esta diferencia puede deberse al hecho de que los parámetros del modelo fueron estimados experimentalmente, lo cual acarrea un error asociado. La probabilidad de que este error influya en el cálculo del tamaño de bloque *TBB* es mayor cuando se emplea un gran número de unidades de procesamiento o cuando las secuencias a alinear son muy grandes. En particular, las diferencias se presentan con el tamaño de secuencia más grande cuando se emplean 32 núcleos y con los dos tamaños de secuencia más grandes cuando se utilizan 64 núcleos. De todas formas, las diferencias entre la mejor eficiencia y la obtenida con el tamaño de bloque *TBB* calculado como óptimo no son significativas, lo que confirma la alta efectividad del modelo desarrollado.

En los Cuadros 7.7 y 7.8 se muestra un resumen de los mejores speedups y de sus correspondientes eficiencias, respectivamente, para los diferentes tamaños de problema y números de unidades de procesamiento seleccionados. Las Figuras 7.5 y 7.6 representan gráficamente a estos dos cuados.

P	***N***					
	65536	**131072**	**262144**	**524288**	**1048576**	**2097512**
16	12,40	12,96	13,27	13,58	13,88	13,94
32	21,10	24,33	25,42	26,27	26,89	27,41
64	35,04	43,28	48,34	50,37	52,93	54,11

Cuadro 7.7: Resumen de los mejores speedups obtenidos con el algoritmo paralelo en las pruebas realizadas en la segunda fase.

P	***N***					
	65536	**131072**	**262144**	**524288**	**1048576**	**2097512**
16	0,77	0,81	0,83	0,85	0,87	0,87
32	0,66	0,76	0,79	0,82	0,84	0,86
64	0,55	0,68	0,76	0,79	0,82	0,84

Cuadro 7.8: Resumen de las mejores eficiencias obtenidas con el algoritmo paralelo en las pruebas realizadas en la segunda fase.

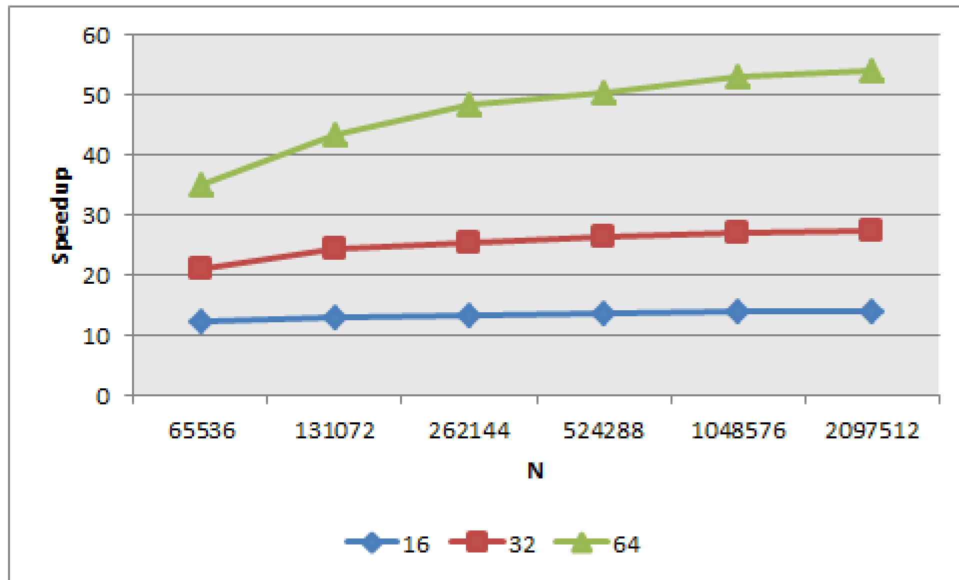

Figura 7.5: Mejores speedups obtenidos por el algoritmo paralelo durante la segunda fase de pruebas.

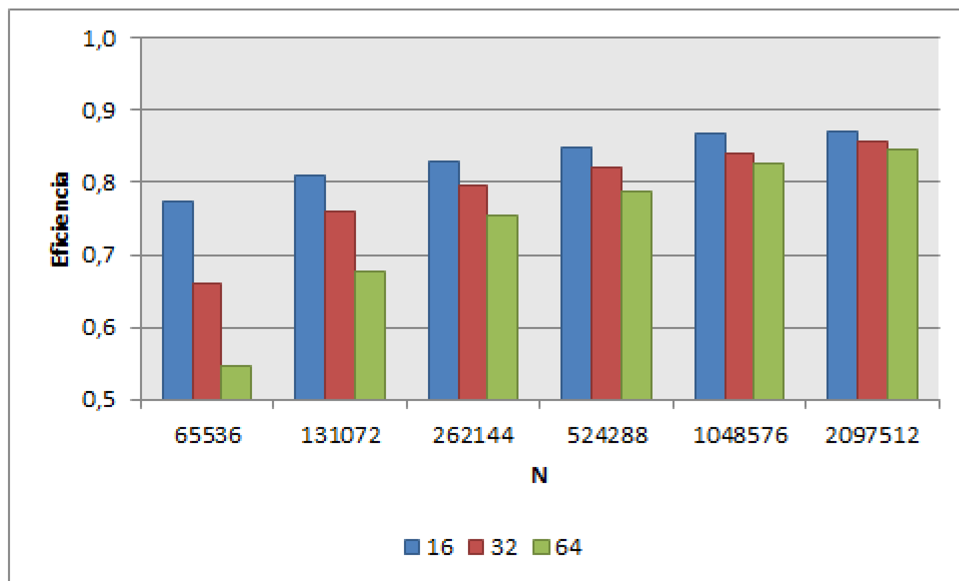

Figura 7.6: Mejores eficiencias obtenidas por el algoritmo paralelo durante la segunda fase de pruebas.

En el gráfico de la Figura 7.5, se puede observar que el speedup crece tanto al aumentar el número de unidades de procesamiento empleadas como el tamaño de problema utilizado. Como en la mayoría de los sistemas paralelos, no ocurre lo mismo con la eficiencia. En el gráfico de la Figura 7.6, se puede apreciar que la eficiencia crece al aumentar el tamaño del problema, pero disminuye al incrementar el número

de unidades de procesamiento. Al utilizar más procesadores, la eficiencia disminuye debido al aumento del overhead (se produce un incremento del ocio así como de la cantidad de comunicaciones presentes en la solución).

Por último, del Cuadro 7.8 y de la Figura 7.6, se puede apreciar que es posible mantener constante la eficiencia obtenida al aumentar tanto el tamaño del problema como el número de unidades de procesamiento. Este comportamiento refleja la característica de escalabilidad presente en este sistema paralelo.

Capítulo 8

Conclusiones y líneas de trabajo futuro

El alineamiento de secuencias de ADN es una de las operaciones centrales de la bioinformática, debido en parte a sus múltiples usos en otras aplicaciones relacionadas a la biología. El algoritmo Smith-Waterman permite realizar alineamientos locales. Por su alto costo computacional, en la práctica se emplean diversas heurísticas que, si bien son más rápidas, no garantizan que el alineamiento óptimo sea encontrado. Aquí radica la importancia de aplicar la potencia de cómputo de plataformas paralelas para acelerar el procesamiento de las secuencias sin perder precisión en los resultados.

Inicialmente se desarrollaron dos algoritmos secuenciales, los cuales aplican distintas técnicas de optimización para resolver el problema. En particular, el segundo algoritmo emplea una técnica de procesamiento por bloques de la matriz que, al maximizar la localidad temporal y espacial de los datos, mejora el rendimiento del primer algoritmo aproximadamente un 5 %.

A continuación, se trabajó en un algoritmo paralelo tomando como arquitectura de soporte un cluster de multicores. Para el mismo, se reutilizaron las técnicas de optimización aplicadas en los algoritmos secuenciales, en particular, la técnica de procesamiento por bloques que mejora la localidad de los datos. Por último, como librería para el pasaje de mensajes se seleccionó MPI teniendo en cuenta su capacidad para lograr alto rendimiento y su alta portabilidad.

Debido a que el máximo rendimiento alcanzable por el algoritmo paralelo desarrollado depende de las características de la arquitectura de soporte y del tamaño de las secuencias a alinear, se desarrolló un modelo que permite configurar la ejecución del mismo para lograr dicho propósito. Se realizaron pruebas variando tanto el tamaño de

las secuencias como la cantidad de núcleos utilizados de la arquitectura de soporte. El análisis presentado sobre los resultados obtenidos en las pruebas realizadas demuestra la efectividad del modelo para configurar la ejecución del algoritmo y el rendimiento eficiente que logra el mismo, más aún si se tiene en cuenta el patrón de interacción empleado (*pipeline*).

A partir de los resultados obtenidos se cree haber aportado una alternativa válida a las heurísticas utilizadas actualmente al momento de realizar alineamientos de secuencias de ADN con la ventaja de no perder precisión en los resultados.

Una línea de trabajo futuro consiste en adaptar el algoritmo paralelo desarrollado para ser ejecutado sobre clusters heterogéneos de multicores y realizar un análisis de rendimiento del mismo. Debido al auge de las placas gráficas como arquitecturas paralelas para resolver problemas de propósito general, otra línea de trabajo futuro consiste en el desarrollo de un algoritmo que se ejecute sobre las mismas.

Referencias

[1] T. K. Atwood and D. J. Parry-Smith, *Introducción a la Bioinformática.* Prentice Hall, 2002.

[2] T. F. Smith and M. S. Waterman, “Identification of common molecular subsequences,” *Journal of Molecular Biology*, vol. 147, pp. 195–197, 1981.

[3] Advanced Micro Devices, Inc. (2009) AMD Multic-core White Paper. [Online]. Available: http://multicore. amd.com/es-ES/AMD-Multi-Core/resources/Technology-Evolution

[4] M. McCool, “Scalable programming models for massively multicore processors,” in *Proceeding of the IEEE*, vol. 96, no. 5, 2007, pp. 816–831.

[5] European Bioinformatics Institute. (2012, 12) What is bioinformatics? [Online]. Available: http://www.ebi.ac.uk/

[6] R. Durbin, S. R. Eddy, A. Krogh, and G. Mitchinson, *Biological Sequence Analysis. Probabilistic models of proteins and nucleic acids.*, 7th ed. Cambridge University Press, 2002.

[7] Cloud Computing and Distributed Systems (CLOUDS) Laboratory. Department of Computer Science and Software Engineering. The University of Melbourne, Australia. (2012, 12) Cluster and grid computing. [Online]. Available: http://ww2.cs.mu.oz.au/678/

[8] Z. Juhász, P. Kacsuk, and D. Kranzlmuller, Eds., *Distributed and Parallel Systems: Cluster and Grid Computing.* Springer Science + Business Media Inc., 2005.

[9] B. Wilkinson and M. Allen, *Parallel Programming. Techniques and Applications Using Networked Workstations and Parallel Computers.*, 2nd ed. Prentice Hall, 2005.

[10] J. Al-Jaroodi, N. Mohamed, H. Jiang, and D. Swanson, "Modeling parallel applications performance on heterogeneous systems," in *Proceedings of the International Parallel and Distributed Processing Symposium 2003*, 2003.

[11] J. Dongarra, I. Foster, G. Fox, W. Gropp, K. Kennedy, L. Torczon, and A. White, *The Sourcebook of Parallel Computing.* Morgan Kauffman, 2003.

[12] K. Olukotun, O. A. Olukotun, L. Hammond, and J. P. Laudon, *Chip Multiprocessors Architecture: Techniques to Improve Throughput and Latency*, M. D. Hill, Ed. Morgan & Claypool, 2007.

[13] T. Rauber and G. Rünger., *Parallel Programming for Multicore and Cluster Systems.* Springer-Verlag Berlin Heidelberg, 2010.

[14] L. Chai, Q. Gago, and D. K. Panda, "Understanding the impact of multi-core architecture in cluster computing: A case study with intel dual-core system," in *Proceedings of the IEEE International Symposium on Cluster Computing and the Grid*, Río de Janeiro (Brazil), 2007.

[15] C. Zhang, X. Yuan, and A. Srinivasan, "Processor Affinity and MPI Performance on SMP-CMP Clusters," in *IEEE International Symposium on Parallel & Distributed Processing, Workshops and Phd Forum (IPDPSW)*, Florida (USA), 2010.

[16] R. Rabenseifner, "Hybrid parallel programming: Performance problems and chances," in *Proceedings of the 45th Cray User Group Conference*, 2003.

[17] E. Rucci, A. De Giusti, F. Chichizola, M. Naiouf, and L. De Giusti, "DNA sequence alignment: hybrid parallel programming on a multicore cluster," in *Recent Advances in Computers, Communications, Applied Social Science and Mathematics*, N. Mastorakis, V. Mladenov, B. Lepadatescu, H. R. Karimi, and C. G. Helmis, Eds., vol. 1, no. 1. Barcelona (Spain): WSEAS Press, September 2011, pp. 183–190.

[18] X. Wu and V. Taylor, "Performance Characteristics of Hybrid MPI/OpenMP Implementations of NAS Parallel Benchmarks SP and BT on Large-scale Multicore Clusters," *The Computer Journal*, vol. 55, pp. 154–167, 2012.

[19] E. Rucci, F. Chichizola, M. Naiouf, , and A. De Giusti, "Performance comparison of parallel programming paradigms on a multicore cluster," in *Proceedings of the*

24th IASTED International Conference on Parallel and Distributed Computing and Systems (PDCS 2012), T. Gonzalez, Ed., vol. 1. Las Vegas (USA): Acta Press, 2012, pp. 216–221.

[20] R. Graham and G. Shipman, "MPI Support for Multi-core Architectures: Optimized Shared Memory Collectives," in *Proceedings of the 15th European PVM/MPI Users' Group Meeting on Recent Advances in Parallel Virtual Machine and Message Passing Interface.* Dublin (Ireland): Springer-Verlag Berlin, Heidelberg, 2008, pp. 130 – 140.

[21] (2012) MPICH. [Online]. Available: www-unix.mcs.anl.gov/mpi/mpich2

[22] (2012) LAM/MPI. [Online]. Available: http://www.lam-mpi.org/

[23] (2012) OpenMPI. [Online]. Available: http://www.open-mpi.org/

[24] A. Grama, G. Karypis, V. Kumar, and A. Gupta, *An Introduction to Parallel Computing. Design and Analysis of Algorithms.*, 2nd ed. Addison Wesley, 2003.

[25] Hewlett-Packard Development Company. HP BladeSystem. [Online]. Available: http://h18004.www1.hp.com/products/blades/compon ents/c-class.html

[26] Hewlett-Packard Development Company . HP BladeSystem c-Class architecture. [Online]. Available: http://h20000.www2.hp.com/bc/docs/support/Support Manual/c00810839/c00810839.pdf

[27] A. Boukerche, A. C. M. A. de Melo, M. Ayala-Rincon, and T. M. Santana, "Parallel smith-waterman algorithm for local dna comparison in a cluster of workstations," in *Experimental and Efficient Algorithms. Proceedings of the 4th International Workshop WEA 2005.*, Santorini Island, Greece, May 10-13 2005.

[28] W. S. Martins, J. B. D. Cuvillo, F. J. Useche, K. B. Theobald, and G. Gao, "A multithreaded parallel implementation of a dynamic programming algorithm for sequence comparison," in *Pacific Symposium on Biocomputing 2001*, 2001.

[29] R. C. F. Melo, M. E. T. Walter, A. C. M. A. Melo, R. Batista, M. Nardelli, T. Martins, and T. Fonseca, "Comparing two long biological sequences using a dsm system," in *Euro-Par 2003 Parallel Processing. Proceedings of the 9th International Euro-Par Conference.*, Klagenfurt, Austria, August 26-29 2003.

[30] S. I. Steinfadt, “Smith-waterman sequence alignment for massively parallel high-performance computing architectures,” Ph.D. dissertation, Kent State University, 2010.

[31] F. Zhang, X.-Z. Qiao, and Z.-Y. Liu, “A parallel smith-waterman algorithm based on divide and conquer,” in *Proceedings of the Fifth International Conference on Algorithms and Architectures for Parallel Processing (ICA3PPí02)*, 2002.

Apéndice A

Estándar Message-Passing Interface (MPI)

A.1. Funciones básicas

En la Figura A.1 se describen las seis funciones MPI básicas [11]. Las etiquetas E y S aplicadas a los parámetros indican cuando una función utiliza pero no modifica un parámetro (E) y cuando no usa un parámetro pero sí lo actualiza (S).

MPI_INIT (`int*argc, char *** argv`)
Inicia una sesión MPI
`argc` y `argv` sólo son necesarios en el lenguaje C, donde representan los argumentos a la función main

MPI_FINALIZE()
Termina la sesión MPI

MPI_COMM_SIZE(`comm`, `size`)
Determina el número de procesos

E	`comm`	Comunicador
S	`size`	Número de procesos en el comunicador `comm`

MPI_COMM_RANK(`comm`, `pid`)
Determina el identificador del proceso actual

E	`comm`	Comunicador
S	`pid`	Identificador del proceso dentro del comunicador `comm`

MPI_SEND(`buf`, `count`, `datatype`, `dest`, `tag`, `comm`)
Envía un mensaje

E	`buf`	Dirección del buffer de envío
E	`count`	Número de elementos a enviar
E	`datatype`	Tipo de datos de los elementos del buffer
E	`dest`	Identificador del proceso que recibirá el mensaje
E	`tag`	Tag del mensaje
E	`comm`	Comunicador

MPI_RECV(`buf`, `count`, `datatype`, `source`, `tag`, `comm`, `status`)
Recibe un mensaje

S	`buf`	Dirección del buffer de recepción
E	`count`	Tamaño del buffer de rececpción (en número de elementos)
E	`datatype`	Tipo de dato de los elementos del buffer de recepción
E	`source`	Identificador del proceso emisor, o MPI_ANY_SOURCE
E	`tag`	Tag del mensaje, o MPI_ANY_TAG
E	`comm`	Comunicador
S	`status`	Objeto Status

Figura A.1: Funciones MPI básicas.

Las funciones MPI_INIT y MPI_FINALIZE se utilizan para iniciar y finalizar una sesión MPI, respectivamente. La función MPI_INIT debe ser invocado antes que cualquier otra función MPI y sólo una única vez por proceso. Luego del llamado a

MPI_FINALIZE, ninguna otra función MPI puede ser invocada.

Las funciones MPI_COMM_SIZE y MPI_COMM_RANK determinan el número de procesos y el identificador asignado al proceso actual dentro del comunicador pasado como argumento, respectivamente.

La función MPI_SEND se emplea para enviar un mensaje y tiene la siguiente forma general:

MPI_SEND(`buf`, `count`, `datatype`, `dest`, `tag`, `comm`)

Esta función especifica que un mensaje de `count` elementos de tipo `datatype` ubicados en `buf` serán enviados al proceso identificado como `dest` dentro del comunicador `comm` con la etiqueta `tag`. Para cada tipo de dato en C, MPI define su propio tipo de datos. En el Cuadro A.1 se muestra dicha correspondencia. En forma similar, MPI también provee nombres para los tipos de datos en Fortran y C++. Además, MPI le permite al programador definir nuevos tipos de datos no contemplados.

Tipo de dato en MPI	**Tipo de dato en C**
MPI_CHAR	signed char
MPI_SHORT	signed short int
MPI_INT	signed int
MPI_LONG	signed long int
MPI_UNSIGNED_CHAR	unsigned char
MPI_UNSIGNED_SHORT	unsigned short int
MPI_UNSIGNED	unsigned
MPI_UNSIGNED_LONG	unsigned long int
MPI_FLOAT	float
MPI_DOUBLE	double
MPI_LONG_DOUBLE	long double
MPI_BYTE	
MPI_PACKED	

Cuadro A.1: Correspondencia entre los tipos de datos soportados por MPI y los correspondientes al lenguaje C.

La función MPI_RECV recibe un mensaje y tiene la siguiente forma general:

MPI_RECV(`buf`, `count`, `datatype`, `source`, `tag`, `comm`, `status`)

El proceso que realiza un llamado a esta función se bloquea a la espera de la recepción de un mensaje con la etiqueta `tag` por parte del proceso identificado como `source` dentro del comunicador `comm`. Cuando el mensaje llega, los elementos de tipo `datatype` son ubicados en el buffer ubicado en `buf`. El programador es el responsable de garantizar que este buffer sea suficientemente grande como para contener al

menos `count` elementos. La variable `status` puede ser utilizada posteriormente para consultar el tamaño, la etiqueta y la fuente del mensaje. Por último, MPI permite en esta función la especificación de argumentos comodines tanto para `source` como para `tag`. Si en `source` se utiliza el comodín MPI_ANY_SOURCE, entonces cualquier proceso dentro del comunicador podrá ser el emisor del mensaje. En forma similar, si en `tag` se usa el comodín MPI_ANY_TAG, entonces pueden ser aceptados mensajes con cualquier etiqueta [11, 24].

A.2. Operaciones de comunicación no bloqueantes

MPI provee variantes no bloqueantes de las funciones de envío y recepción: MPI_ISEND y MPI_IRECV. Los llamados a estas funciones no esperan a que la comunicación se complete para regresar. Para poder verificar si una operación se completó, MPI cuenta con la función MPI_WAIT. En la Figura A.2 se describen estas funciones.

MPI_ISEND(`buf`, `count`, `datatype`, `dest`, `tag`, `comm`, `request`)
Inicia el envío de un mensaje

E	`buf`	Dirección del buffer de envío
E	`count`	Número de elementos a enviar
E	`datatype`	Tipo de datos de los elementos del buffer
E	`dest`	Identificador del proceso que recibirá el mensaje
E	`tag`	Tag del mensaje
E	`comm`	Comunicador
S	`request`	Objeto Request

MPI_IRECV(`buf`, `count`, `datatype`, `source`, `tag`, `comm`, `request`)
Inicia la recepción de un mensaje

S	`buf`	Dirección del buffer de recepción
E	`count`	Tamaño del buffer de rececpción (en número de elementos)
E	`datatype`	Tipo de dato de los elementos del buffer de recepción
E	`source`	Identificador del proceso emisor, o MPI_ANY_SOURCE
E	`tag`	Tag del mensaje, o MPI_ANY_TAG
E	`comm`	Comunicador
S	`request`	Objeto Request

MPI_WAIT(`request`, `status`)
Espera a que la operación especificada en el objeto Request se complete

S	`request`	Objeto Request
S	`status`	Objeto Status

Figura A.2: Funciones MPI para comunicaciones no bloqueantes.

A.3. Operaciones de comunicación colectivas

MPI provee un conjunto extensivo de funciones que realizan operaciones de comunicación colectivas comúnmente usadas. Todas estas funciones toman como argumento un comunicador que define el grupo de procesos involucrados en la operación de comunicación. Todos los procesos pertenecientes a este comunicador participan en la operación y es necesario que todos invoquen a la función correspondiente [24]. En la Figura A.3 se describen las funciones MPI más relevantes para comunicaciones colectivas.

La función MPI_BARRIER se utiliza para sincronizar procesos.

La función MPI_BCAST se emplea para enviar datos de un proceso a todos los restantes. Esta función envía `count` elementos de tipo `datatype` almacenados en `buf` del proceso `root` a todos los demás procesos dentro del comunicador `comm`.

MPI_BARRIER(`comm`)

Sincroniza todos los procesos

E	`comm`	Comunicador

MPI_BCAST(`buf`, `count`, `datatype`, `root`, `comm`)

Envía datos de un proceso a todos los demás

E/S	`buf`	Dirección del buffer de envío/recepción
E	`count`	Tamaño del buffer de envío/recepción (en número de elementos)
E	`datatype`	Tipo de dato de los elementos del buffer
E	`root`	Identificador del proceso emisor
E	`comm`	Comunicador

MPI_GATHER(`sendbuf`, `sendcount`, `sendtype`, `recvbuf`, `recvcount`, `recvtype`, `root`, `comm`)

Agrupa datos de todos los procesos en un único proceso

E	`sendbuf`	Dirección del buffer de envío
E	`sendcount`	Número de elementos que cada proceso envía
E	`sendtype`	Tipo de dato de los elementos del buffer de envío
S	`recvbuf`	Dirección del buffer de recepción (significativo sólo para proceso `root`)
E	`recvcount`	Número de elementos a recibir de cada proceso (significativo sólo para proceso `root`)
E	`recvtype`	Tipo de dato de los elementos del buffer de recepción (significativo sólo para proceso `root`)
E	`root`	Identificador del proceso receptor
E	`comm`	Comunicador

MPI_SCATTER(`sendbuf`, `sendcount`, `sendtype`, `recvbuf`, `recvcount`, `recvtype`, `root`, `comm`)

Reparte datos de un proceso a todos los demás

E	`sendbuf`	Dirección del buffer de envío (significativo sólo para proceso `root`)
E	`sendcount`	Número de elementos a enviar a cada proceso (significativo sólo para proceso `root`)
E	`sendtype`	Tipo de dato de los elementos del buffer de envío (significativo sólo para proceso `root`)
S	`recvbuf`	Dirección del buffer de recepción
E	`recvcount`	Número de elementos a recibir por cada proceso
E	`recvtype`	Tipo de dato de los elementos del buffer de recepción
E	`root`	Identificador del proceso emisor
E	`comm`	Comunicador

MPI_REDUCE(`sendbuf`, `recvcount`, `count`, `type`, `op`, `root`, `comm`)

Realiza una operación de reducción todos a uno.

E	`sendbuf`	Dirección del buffer de envío
S	`recvbuf`	Dirección del buffer de recepción (significativo sólo para proceso `root`)
E	`count`	Número de elementos del buffer de envío
E	`type`	Tipo de dato de los elementos del buffer de envío
E	`op`	Operación de reducción
E	`root`	Identificador del proceso receptor
E	`comm`	Comunicador

Figura A.3: Funciones MPI más relevantes para comunicaciones colectivas.

La función MPI_GATHER le permite a un proceso agrupar datos de todos los procesos. Cada proceso, incluido el proceso `root`, envía `sendcount` elementos de tipo `sendtype`, los cuales se encuentran almacenados en `sendbuf`. Como resultado, el proceso `root` recibirá p buffers de `recvcount` elementos de tipo `recvtype` (siendo p el número de procesos formados por el comunicador `comm`). Los datos recibidos son almacenados en `recvbuf`, ordenados de acuerdo a los identificadores de los procesos.

La función MPI_SCATTER le permite a un proceso repartir datos entre todos los procesos. El proceso `root` envía diferentes bloques del buffer almacenado en `sendbuf` a todos los procesos dentro del comunicador `comm` (incluido a sí mismo) ordenados de acuerdo a los identificadores de los mismos. Cada proceso recibe `sendcount` elementos contiguos de tipo `sendtype`.

La función MPI_REDUCE permite que un conjunto de procesos realice una reducción todos a uno. Esta función combina los elementos almacenados en el buffer `sendbuf` de cada proceso dentro del comunicador `comm`, usando la operación `op`, y guarda el resultado en el buffer `recvbuf` del proceso `root`. En el Cuadro A.2 se muestran las operaciones de reducción predeterminadas que provee MPI [24].

MPI_Op	**Operación**
MPI_MAX	Máximo
MPI_MIN	Mínimo
MPI_SUM	Suma
MPI_PROD	Producto
MPI_LAND	AND lógico
MPI_BAND	AND bit a bit
MPI_LOR	OR lógico
MPI_BOR	OR bit a bit
MPI_LXOR	XOR lógico
MPI_BXOR	XOR bit a bit
MPI_MAXLOC	Máximo y su correspondiente índice
MPI_MINLOC	Mínimo y su correspondiente índice

Cuadro A.2: Operaciones de reducción en MPI

Apéndice B

Estimación de parámetros del modelo para el cálculo del tamaño *TBB* óptimo en la arquitectura de soporte

B.1. Parámetros t_s y t_b

Si bien todas las unidades de procesamiento de la arquitectura de soporte poseen la misma potencia de cómputo, la latencia y el ancho de banda de comunicación entre núcleos varía de acuerdo a la ubicación de los mismos como se describió en la Sección 2.2. Estimar los valores t_s y t_b relacionados a la arquitectura teniendo en cuenta un enlace de comunicación que no es el más lento, provocaría un solapamiento inadecuado en la solución paralela, ya que los procesos que se comunican por enlaces más rápidos tendrían que esperar por aquellos que se comunican por enlaces más lentos. Es por ello que la estimación de los valores t_s y t_b se realizó utilizando la red de interconexión que comunica a núcleos que se encuentran en diferentes nodos.

Para las pruebas se utilizaron dos procesos MPI que intercambian mensajes (envían y reciben alternadamente) en forma repetida. El primer proceso comienza enviando y luego recibe, mientras que el segundo proceso empieza recibiendo y luego envía. Los procesos se mapearon a nodos diferentes de la arquitectura de soporte y se usaron las funciones MPI_Isend() y MPI_Recv() para el envío y la recepción de mensajes, respectivamente. Para el parámetro t_s se intercambiaron mensajes vacíos (parámetro `count` de la función MPI_Isend igual a cero). En el caso del parámetro t_b, se utilizaron mensajes de diferentes tamaños (`count` $= \{8, 16, 32, 48, 64, 80, 96, 112, 128\}$). Además, una vez calculado el valor de t_s, se restó el mismo a los valores

obtenidos para t_b. Cada proceso envió (y recibió) 5000000 de mensajes. Este número se seleccionó de manera de garantizar que cada prueba particular se ejecutara durante al menos cinco minutos. Por último, cada instancia de prueba se repitió diez veces calculando el tiempo de ejecución promedio para el análisis. En el Cuadro B.1 se muestra un esquema representativo de las pruebas realizadas para la estimación de los parámetros t_s y t_b junto a sus valores finales.

	t_s	t_b								
Número de mensajes enviados por proceso	5000000									
Tamaño del mensaje	0	8	16	32	48	64	80	96	112	128
Promedio final	0,00005320314865	0,00000010869568								

Table B.1: Esquema representativo de las pruebas realizadas para la estimación de los parámetros t_s y t_b junto a sus valores finales.

B.2. Parámetro t_c

Para la estimación del parámetro t_c se utilizaron algunos de los resultados obtenidos en la primera fase de pruebas descrita en la Subsección 6.3.1. Se seleccionaron los tiempos de ejecución obtenidos por el algoritmo secuencial por bloques para los diferentes tamaños de problema utilizando el tamaño de bloque *TBA* identificado como óptimo como se indica en la Sección 7.1. A cada tiempo de ejecución particular se lo dividió por la cantidad de elementos de la correspondiente matriz construida y luego se calculó el promedio de todos ellos. En la Tabla B.2 se muestra un esquema representativo de las pruebas realizadas para la estimación del parámetro t_c junto a su valor final.

	t_c					
N	65536	131072	262144	524288	1048576	2097512
TBA	16384					
Promedio final	0,00000002891760					

Table B.2: Esquema representativo de las pruebas realizadas para la estimación del parámetro t_c junto a su valor final.

Apéndice C

Trabajos relacionados

La paralelización del alineamiento de secuencias de ADN mediante el método Smith-Waterman resulta ser un tema ampliamente estudiado por la comunidad científica [27, 28, 29, 30, 31]. Sin embargo, los trabajos encontrados sobre este tema suelen realizar relajaciones al algoritmo original para facilitar el procesamiento o resuelven el alineamiento en forma completa, es decir, no sólo calculan el puntaje de similitud sino que también buscan el alineamiento óptimo. Estas dos condiciones no permiten realizar comparaciones con éste trabajo en forma directa.