Universidad de Buenos Aires
Facultad de Ciencias Exactas y Naturales
Departamento de Computación

Optimización de la resolución numérica de una teoría molecular

Tesis presentada para optar al título de
Licenciado en Ciencias de la Computación

Ignacio Gleria

Director: Dr. Esteban Mocskos
Codirector: Dr. Mario Tagliazucchi
Buenos Aires, 2020

OPTIMIZACIÓN DE LA RESOLUCIÓN NUMÉRICA DE UNA TEORÍA MOLECULAR

La Teoría Molecular es una metodología teórico-computacional para el estudio de materiales blandos (polímeros, surfactantes, geles, membranas biológicas, etc.). Por ejemplo, esta teoría se ha usado para estudiar procesos de autoensamblado de polímeros, nanoporos biológicos y nanopartículas modificadas por polímeros. Brevemente, el objetivo de la teoría es encontrar, para cada posición dentro del sistema, las densidades y estados químicos de cada molécula presente (por ej. un polímero, moléculas de solventes y/o iones). Para ello, se escribe la energía libre del sistema (en forma aproximada) como un funcional de estas funciones desconocidas. Al minimizar analíticamente el funcional de energía libre, se obtienen expresiones analíticas para estas funciones desconocidas. Estas expresiones se discretizan siguiendo un esquema de diferencias finitas, lo cual resulta en un sistema de ecuaciones no-lineales acopladas. Resolver estas ecuaciones involucra, entonces, encontrar un vector de soluciones $\mathbf{x}$ (el cual contiene una solución para las densidades y estados químicos), tal que $\mathbf{F}(\mathbf{x}) = 0$ (donde $\mathbf{F}$ es el conjunto de ecuaciones no-lineales acopladas). La resolución de este problema se realiza empleando una variante del método de Newton. La evaluación de $\mathbf{F}(\mathbf{x})$ en cada iteración del método de Newton es el paso computacionalmente más costoso de la resolución, dado que requiere la multiplicación y suma de matrices dispersas (también conocidas como matrices ralas o esparzas en castellano y *"sparse matrices"* en inglés) de gran tamaño. Actualmente, el almacenamiento y cómputo de estas matrices ralas se realiza empleando un formato comprimido (es decir, un formato en el cual se eliminan los elementos nulos de las matrices), pero la eficiencia de las rutinas empleadas para multiplicar y sumar las matrices no ha sido evaluada críticamente y ni optimizada. Además, el código del que se parte se encuentra paralelizado para múltiples procesadores empleando el estándar MPI, pero no está optimizado para arquitecturas masivamente paralelas.

Palabras claves: Nanoporos, MPI, CPU, Paralelismo, Memoria Compartida.

OPTIMIZATION OF THE NUMERICAL RESOLUTION OF A MOLECULAR THEORY

The Molecular Theory is a theoretical and computational framework for the study of soft matter (polymers, surfactants, gels, biological membranes, etc). For example, this theory has been used to study polymer self-assembly, biological nanopores and nanoparticles modified by polymers. The objective of this framework is to find the density and chemical state of all molecules (polymers, solvent molecules and/or ions) in each position of the system. To achieve this, the system's free energy is written down as an approximated functional of these unknown functions. Analytically minimizing the free energy functional provides analytical expressions for the densities and chemical states. These expressions are discretized following a finite differences scheme which results in a coupled non linear equation system. In order to find a solution for this system we have to find a solution vector $\mathbf{X}$ such as $\mathbf{F}(\mathbf{x}) = 0$ (where $\mathbf{F}$ is the set of coupled non linear equations). The solution to this problem is obtained by employing a variant of Newton's method. The evaluation of $\mathbf{F}(\mathbf{x})$ in each iteration of the method is the most taxing part of the whole program (performance wise) because it requires the multiplication and addition of very large sparse matrices. As of today, the storage and computation of these sparse matrixes operations is done employing an in-house compressed format which doesn't store the null elements, but the real efficiency of the employed routines hasn't been evaluated for possible optimizations. Additionally, the code employs the MPI standard for parallelizing the computations, but doesn't implement optimizations for massively parallel architectures.

Keywords: Nanopores, MPI, CPU, paralelization, OpenMP.

AGRADECIMIENTOS

Al amor de mi vida, Cons. Sos todo. A mamá, papá y Caro, por tenerme fe y apoyarme siempre. A Pablo, Adriana, George, Nat y Lilith , por aceptarme en sus familias. A la bobe y al zeide, por (sin haber podido verlo) permitirme conocer Israel y Europa. Al resto de mi familia, por darme tantos sobrinitos y sobrinitas, por tantas alegrías compartidas. A mis amigues del Pelle, ya saben quiénes son. A mis amigues de la carrera, gracias por tantos mates compartidos, trabajos entregados, cursadas sufridas y sobre todo risas. A los litos, Andy, Yani, Marito, Santi: ya no me van a tener que aguantar llorando por la tesis (obvio que siempre voy a tener una excusa para quejarme). A Esteban y Mario, por haberme tenido infinita paciencia con esta tesis: gracias. A los jurados, por haberse tomado el tiempo de leer esta tesis. A Julio Augusto Mascitti y al Centro de Simulación Computacional para Aplicaciones Tecnológicas por proveerme acceso a TUPAC para los experimentos finales. A la Universidad de Buenos Aires y a la Facultad de Ciencias Exactas y Naturales, por abrirme la mente y darme las herramientas necesarias para estar donde estoy hoy.

A todes, GRACIAS.

Por una educación pública, gratuita, laica y de calidad

Índice general

1. INTRODUCCIÓN

1.1. La Teoría Molecular

La teoría molecular[1, 2] es una herramienta mecánico-estadística para el modelado de sistemas de materia blanda. Estos sistemas se caracterizan por cambiar su estructura fácilmente en respuesta a estímulos físicos o químicos débiles. Ejemplos típicos incluyen sistemas poliméricos, surfactantes y geles. El modelado de estos sistemas puede realizarse empleando diversas estrategias, las que pueden considerar distintos grados de aproximación. Las técnicas basadas en la mecánica cuántica proporcionan una descripción detallada de la estructura electrónica del sistema, pero resultan extremadamente costosas para la escala de estos sistemas, con lo que casi no se las emplean en estos escenarios. Las simulaciones de dinámica molecular atomística clásica dan una descripción de la evolución temporal de cada átomo del sistema a partir de las leyes de Newton [3]. A pesar de tener un menor grado de detalle a nivel molecular, también resultan muy costosas debido al tamaño de los sistemas y las escalas de tiempo involucrados.

Una aproximación muy usada es agrupar conjuntos de átomos, representándolos con una nueva partícula (generalmente denominada *bead*). Esta aproximación se conoce como aproximación de **grano grueso** (*coarse grain*) y es muy utilizada en simulaciones de materia blanda. En el extremo opuesto a las técnicas de simulación, tenemos las teorías analíticas (por ejemplo los llamados argumentos de escala [4]). Éstas son teorías que, en general, parten de una expresión para la energía libre del sistema que es minimizada para obtener el estado de equilibrio. Si bien conllevan un costo comunicacional bajo o nulo, deben involucrar aproximaciones que, generalmente, hacen perder detalles moleculares del sistema.

La idea de la **teoría molecular** es proveer una descripción aproximada del sistema que sea computacionalmente más accesible que las simulaciones de dinámica molecular, pero que incorpore en forma explícita detalles moleculares del sistema. Estos detalles incluyen la forma, tamaño, carga y conformaciones (es decir grados de libertad internos) de todas las moléculas. Además, se consideran las distintas interacciones electrostáticas y no electrostáticas entre estas moléculas, así como también la presencia de equilibrios químicos (por ejemplo equilibrios ácido/base o ligando/receptor). El hecho de que la teoría molecular parta de una expresión para la energía libre del sistema permite obtener cantidades termodinámicas que son, generalmente, costosas de calcular para los métodos de simulación. La teoría molecular permite, además, aprovechar la simetría del sistema para reducir el costo del cálculo comunicacional. Es importante notar que la derivación de la teoría molecular realiza varias aproximaciones, las cuales deben siempre tenerse en cuenta al estudiar sistemas puntuales. Además, al incorporar gran grado de detalle molecular, la teoría molecular no puede ser resuelta en forma analítica y, por lo tanto, debe resolverse numéricamente. Sin embargo, en los modelos más simples la resolución numérica también es aplicada, ya que es intrínseca al método.

El grupo de materiales blandos de la FCEN-UBA[1] emplea el método de la teoría molecular en la actualidad para estudiar distintos problemas de interés, tales como autoensamblados de surfactantes [5, 6], adsorción de proteínas [7], nanocanales modificados por

[1] `http://www.inquimae.fcen.uba.ar/softmaterials/`

polímeros [8] y películas poliméricas en superficies [9, 10]. En todos los casos, las aplicaciones numéricas que resuelven los distintos escenarios estudiados con la teoría molecular han sido desarrollados íntegramente en el grupo.

1.1.1. Formulación de la teoría

El sistema que modelamos (ver figura 1.1) es un nanoporo corto de radio R y longitud L. La superficie interna de la membrana está modificada mediante N_P cadenas de polímeros ancladas en ella, con una densidad superficial de $\frac{N_P}{A}$. Adicionalmente, cada cadena de polímeros contiene N monómeros.

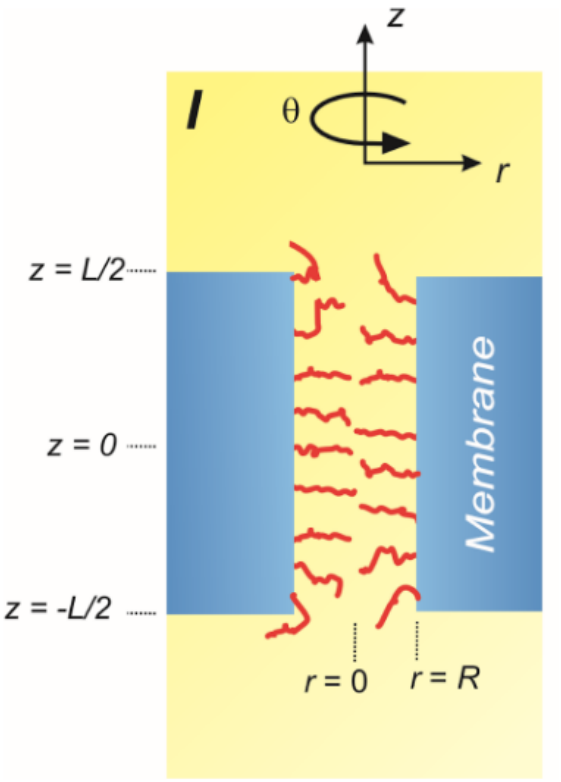

Fig. 1.1: Representación esquemática de un nanoporo cilíndrico, incluyendo el sistema de coordenadas que lo describe.

Con el objetivo de describir nuestro marco teórico, se propone un funcional de energía libre F para el sistema, dado por la siguiente fórmula:

$$\begin{aligned} \beta F = & \int \rho_s(\mathbf{r}) \left[\ln\left(\rho_s(\mathbf{r}) v_s\right) - 1\right] d\mathbf{r} \\ & + \frac{N_P}{A} \int \sum_{\alpha} P_P(\mathbf{r}(\mathbf{s}), \alpha) \ln\left(P_P(\mathbf{r}(\mathbf{s}), \alpha)\right) d\mathbf{s} \\ & + \frac{\beta\chi}{2} \iint g\left(\left|\mathbf{r} - \mathbf{r}'\right|\right) \left\langle n_P(\mathbf{r})\right\rangle \left\langle n_P\left(\mathbf{r}'\right)\right\rangle d\mathbf{r} d\mathbf{r}' \end{aligned} \tag{1.1}$$

donde $\beta = 1/kT$ y $\mathbf{r}$ es el vector posición, $\mathbf{r} = (\mathrm{x}, \mathrm{y}, \mathrm{z})$.

- El primer término de la ecuación es la entropía translacional de las moléculas del solvente, siendo $\rho_{\mathrm{s}}(\mathbf{r})$ su densidad en $\mathbf{r}$.
- El siguiente término es la entropía conformacional de las cadenas de polielectrolitos, donde $\mathbf{s}$ es una parametrización del área de la membrana que contiene polímeros anclados, es decir el área interna del nanocanal, siendo $d\mathbf{s}$ el elemento de área y $P_P(\mathbf{r}(\mathbf{s}), \alpha)$ es la probabilidad de tener una cadena anclada en el punto $\mathbf{r}(\mathbf{s})$ de la conformación α.
- El último término es la energía efectiva de atracción de van de Waals (vdW) entre segmentos [1], que representa la diferencia entre la energía de atracción segmento-segmento y la energía de atracción segmento-solvente. En este término, χ determina la fuerza de la interacción, mientras que $g\left(\left|\mathbf{r} - \mathbf{r}'\right|\right)$ es una función atractiva vdW dependiente de la distancia y con la siguiente forma:

$$\begin{aligned} & g\left(\left|\mathbf{r} - \mathbf{r}'\right|\right) = -\left(\tfrac{a}{\left|\mathbf{r} - \mathbf{r}'\right|}\right)^6 && \text{para } a < \left|\mathbf{r} - \mathbf{r}'\right| < 1{,}5\delta \\ & g\left(\left|\mathbf{r} - \mathbf{r}'\right|\right) = 0 && \text{en otro caso} \end{aligned} \tag{1.2}$$

  donde a es la longitud del segmento y $1{,}5\delta$ es un parámetro de corte. $\langle n_P(\mathbf{r})\rangle$ es la densidad de segmentos de polímero en $\mathbf{r}$:

$$\left\langle n_P(\mathbf{r})\right\rangle = \frac{N_P}{A} \int \sum_{\alpha} P_P\left(\mathbf{r}'\left(\mathbf{s}'\right), \alpha\right) n_P\left(\mathbf{r}'\left(\mathbf{s}'\right), \alpha, \mathbf{r}\right) d\mathbf{s}' \tag{1.3}$$

  donde $n_P\left(\mathbf{r}'\left(\mathbf{s}'\right), \alpha, \mathbf{r}\right) \mathrm{d}\mathbf{r}$ es el número de segmentos que una cadena anclada en $\mathbf{r}'\left(\mathbf{s}'\right)$ tiene en el volumen entre $\mathbf{r}$ y $\mathbf{r} + d\mathbf{r}$ cuando está en la conformación α. $d\mathbf{s}'$ es el elemento de área, mientras que la integral abarca el área donde se encuentran los polímeros anclados.

Las repulsiones moleculares son tratadas en este modelo como interacciones de volumen excluido y modeladas como una restricción de empaquetamiento de la siguiente forma para cada $\mathbf{r}$:

$$\rho_s(\mathbf{r}) v_s + \left\langle \phi_P(\mathbf{r})\right\rangle = 1 \tag{1.4}$$

donde la fracción total de volumen del polímero, $\langle \phi_p(\mathbf{r})\rangle$, es dada por

$$\left\langle \phi_P(\mathbf{r})\right\rangle = \left\langle n_P(\mathbf{r})\right\rangle v_P \tag{1.5}$$

donde v_P es el volumen de un segmento de un polímero.

La restricción de empaquetamiento es aplicada cuando se minimiza la ecuación 1.1 introduciendo un multiplicador de Lagrange $\beta\pi(\mathbf{r})$ (que es la presión osmótica dependiente de la posición [1, 2]). Más específicamente, minimizamos:

$$\beta \mathrm{W} = \beta F + \int \beta\pi(\mathbf{r}) \left[\langle \phi_P(\mathbf{r}) \rangle + \rho_s(\mathbf{r}) v_s - 1\right] d\mathbf{r} \tag{1.6}$$

con respecto a las funciones $\rho_\mathrm{s}(\mathrm{r})$ y $P_P(\mathbf{r}, \alpha)$. La derivada funcional respecto a $\rho_\mathrm{s}(\mathbf{r})$ produce la siguiente relación:

$$\rho_s(\mathbf{r}) v_s = \exp\left(-v_s \beta\pi(\mathbf{r})\right) \tag{1.7}$$

La variación con respecto a $P_P(\mathbf{r}, \alpha)$ nos lleva a la función de distribución de probabilidades para las cadenas de polímeros:

$$P_P\left(\mathbf{r}', \alpha\right) = \quad \frac{1}{\xi\left(\mathbf{r}'\right)} \exp\left\{ - \int n_P\left(\mathbf{r}', \alpha, \mathbf{r}\right) \left[\beta v_P \pi(\mathbf{r}) + \int \beta\chi g\left(\left|\mathbf{r} - \mathbf{r}''\right|\right) \left\langle n_P\left(\mathbf{r}''\right)\right\rangle d\mathbf{r}'' \right] d\mathbf{r} \right\} \tag{1.8}$$

donde $\xi\left(\mathrm{r}'\right)$ es una constante de normalización con $\Sigma_\alpha P_p\left(\mathbf{r}', \alpha\right) = 1$

1.1.2. Discretización y solución numérica

Una ventaja de la teoría molecular respecto de otras técnicas de modelado, como por ej. simulaciones de dinámica molecular, es que permite aprovechar la simetría del sistema. En el caso del sistema en la figura 1.1, resulta natural considerar un sistema de coordenadas cilíndrico, dado por las coordenadas axial (z), radial (r) y angular (θ). Para el caso de un nanoporo corto conectando dos reservorios, es posible suponer que el sistema es homogéneo en la coordenada angular y por lo tanto las ecuaciones dependen solo de z y r (aproximación por simetría axial).

Las ecuaciones de la teoría molecular se reescriben en términos de las coordenadas z y r, y luego las ecuaciones 1.7 y 1.8 son sustituidas en la restricción de empaquetamiento 1.4. La ecuación resultante es discretizada en las coordenadas r y z utilizando celdas de dimensión $\delta \times \delta$, con $\delta = 0{,}5$ nm. El número de celdas en r y z son M_r y M_z respectivamente. Un detalle a tener en cuenta es que mientras el sistema es en principio infinito debido al tamaño infinito de los reservorios, solamente hace falta utilizar M_r y M_z lo suficientemente grandes como para contener todas las posibles cadenas de conformaciones[11].

Las ecuaciones obtenidas al minimizar la energía libre son discretizadas reemplazando las integrales por sumas. Por ejemplo, discretizar la ecuación 1.4 resulta en la siguiente fórmula:

$$\rho_s\left(j_r, j_z\right) v_s + \left\langle \phi_P\left(j_r, j_z\right)\right\rangle = 1 \tag{1.9}$$

En esta ecuación los índices discretos j_r y j_z denotan la sección cilíndrica con radio interno $r = \left(j_r - 1\right)\delta$, radio externo $r = j_r\delta$ y una delimitación dada por los planos $z = \left(j_z - 1\right)\delta$ y $z = j_z\delta$. Notar que solamente las celdas que no son parte de la membrana deberían ser consideradas para la ecuación 1.9.

Para resolver numéricamente la teoría molecular, las expresiones 1.3 y 1.8 se reemplazan en la restricción de empaquetamiento (ecuación 1.4). La función $\pi(\mathbf{r})$ requerida en la ecuación 1.8 puede obtenerse a partir de la ecuación 1.7. De esta forma, la única incógnita remanente es la función $\rho_s(\mathbf{r})$.

La restricción de empaquetamiento (ecuación 1.4) discretizada e igualada a cero constituye el sistema de ecuaciones $F(\mathbf{x})$, donde $\mathbf{x}$ es el vector resultante de discretizar $\rho_s(\mathbf{r}) * v_s$ (es decir, la fracción de volumen del solvente en cada punto del espacio).

1.1.3. Modelo molecular

Las sumas sobre todas las posibles conformaciones de polímeros (por ej. suma en α en 1.3) son aproximadas en la teoría por una suma sobre un conjunto representativo de 2×10^5 conformaciones. Estas conformaciones son generadas utilizando un modelo de rotación isomérica [12], con una longitud de segmento de 0.5 nm. Únicamente las conformaciones que se evitan entre sí y que no se sobreponen en la membrana son tenidas en cuenta. Para prevenir un sesgo hacia direcciones específicas, cada secuencia en la unión de las conformaciones es rotada aleatoriamente utilizando 36 ángulos de Euler [1] elegidos al azar. En tanto, el volumen molecular de segmento es de $0.095\,\mathrm{nm}^3$ mientras que el de solvente es de $0.03\,\mathrm{nm}^3$.

1.2. Conceptos computacionales relevantes

En las ciencias de la computación, a lo largo de las décadas ha habido una permanente búsqueda de aprovechar al máximo la capacidad de cómputo de los dispositivos que estuvieran disponibles en cada momento, al mismo tiempo que se fueron acumulando los avances a nivel hardware de éstos. Podemos establecer el inicio de esta maratón tecnológica con el lanzamiento del primer microprocesador comercial, el Intel 4004 [13]; a lo largo de los años los avances en capacidad de cómputo serían vertiginosos.

1.2.1. La Ley de Moore

Una estimación de lo que depararía el futuro de los microprocesadores en cuanto a performance fue la famosa ley de Moore, que a pesar de no ajustarse tanto a los avances de los últimos años[14], fue durante décadas un punto de referencia para los fabricantes de procesadores a nivel mundial. Dicha ley establece que la cantidad de transistores (el elemento más básico que posee un procesador y que forma parte esencial de su diseño) se duplicaría cada 18 meses [14].

Como se puede apreciar en la Figura 1.2, la ley de Moore parecía una profecía que deparaba crecimiento en rendimiento computacional por décadas, pero a partir del año 2004 esta profecía pareció llegar a su fin. Las limitaciones físicas y químicas en la fabricación de microprocesadores implicaron que no se pudiera incrementar la velocidad de reloj de los mismos, ya que problemas en la disipación del calor provocan inestabilidad en el funcionamiento de los mismos [15]. Esto implicó que hubiera una notable disminución en los incrementos de velocidad de reloj (ver nuevamente la Figura 1.2). Sin embargo, la cantidad de transistores se siguió incrementando, pero con otros objetivos que describiremos a continuación.

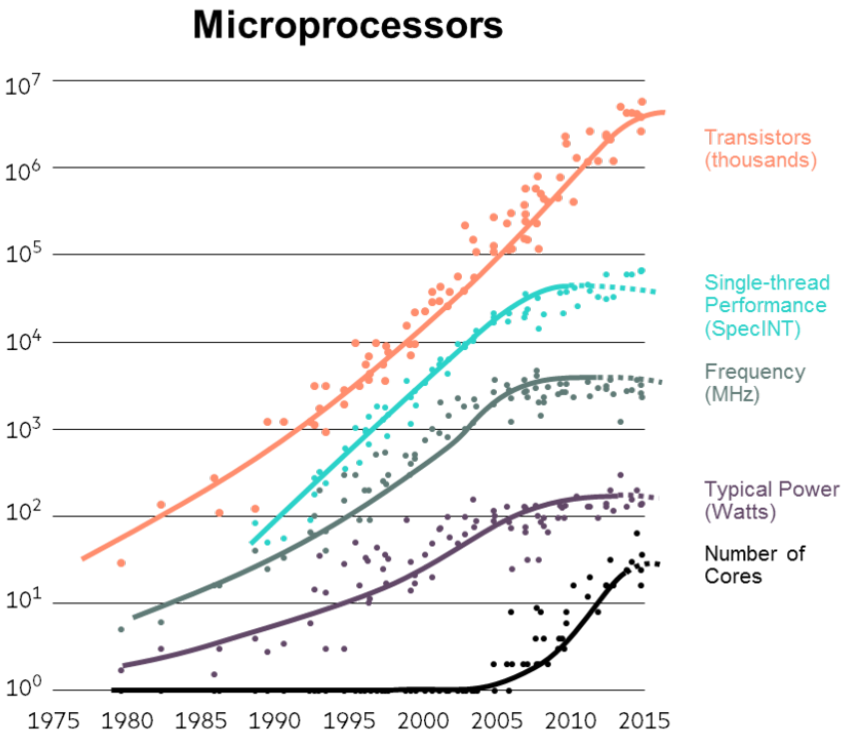

Fig. 1.2: Evolución de distintas características de los microprocesadores a medida que las nuevas generaciones fueron lanzados comercialmente. Gráfico de `https://www.karlrupp.net/2015/06/40-years-of-microprocessor-trend-data/`

1.2.2. Avances tecnológicos relevantes en procesadores

El incremento de la cantidad de transistores por chip a lo largo de las décadas permitió ir agregando diversas características con el objetivo de aumentar la cantidad de operaciones por segundo (a partir de ahora, *throughput*) que pueden realizar. Las características relevantes para este trabajo son:

Vectorización y SIMD

Inicialmente las computadoras procesaban un dato por cada instrucción, lo que se denomina **SISD** (*Single Instruction Single Data*) [16]. A partir de los años 70, las supercomputadoras empezaron a implementar un modelo **SIMD** (*Single Instruction, Multiple Data*) [16], para el cual se contaba con registros especiales capaces de guardar datos contiguos y así realizar cuentas en paralelo. Un ejemplo de este comportamiento puede verse en la Figura 1.3, en la cual podemos observar tres registros de un procesador que tienen un tama no de 256 bits. Estos registros almacenan ocho números de 32 bits, y mediante una instrucción **SIMD** de adición se los suma uno contra uno de manera simultánea (es decir, una suma de vectores), almacenando el resultado en un tercer registro y así logrando realizar ocho sumas por ciclo de procesador en lugar de una.

En la década de los 90 en adelante, este modelo **SIMD** empezó a llegar a los procesadores de uso hogareño cuyas implementaciones iniciales serían `MMX` [13] por parte de Intel y `3DNow!` [17] por parte de AMD.

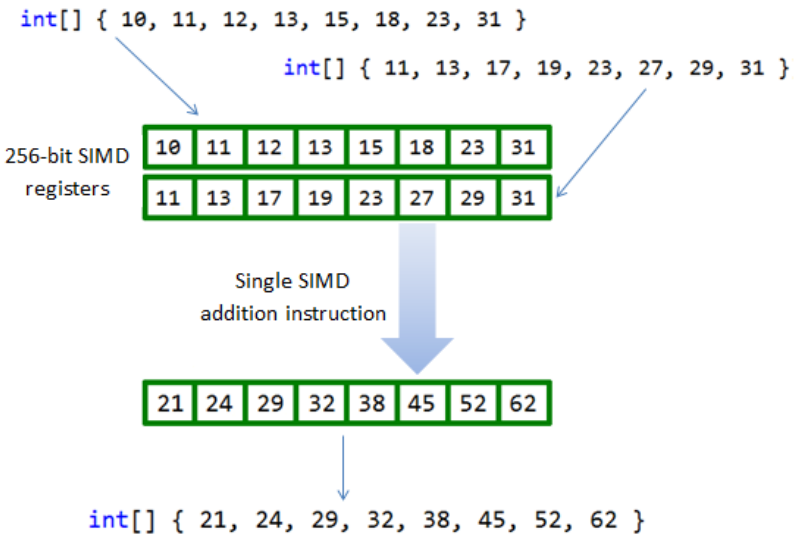

Fig. 1.3: Ejemplo de una suma SIMD

Un punto crucial para que un programa pueda aprovechar las capacidades SIMD de un procesador es que el código sea *vectorizable*, es decir, que se pueda operar con varios elementos simultáneamente. Para que una porción de código sea vectorizable, todas las operaciones no deben tener dependencias entre sí. Para que una operación **S** tenga una dependencia de una operación **T**, deben cumplirse simultáneamente las siguientes propiedades:

- **T** es ejecutado antes que **S**.
- **S** y **T** acceden al mismo dato.
- Al menos uno de los accesos es una escritura.

Algoritmo 1 Ejemplo de un programa que no es vectorizable debido a dependencias entre instrucciones. Dependencia tipo RAW (*Read After Write*).

```
for i ← 1; i < MAX ; i++ do
    a[i] ← a[i-1] + b[i]
end for
```

Algoritmo 2 Ejemplo de código no vectorizable con una dependecia tipo WAW (*Write After Write*).

```
for i ← 1; i < 5 ; i++ do
    acum ← a[i-1] + b[i]
end for
```

Cuando estamos ante una dependencia de datos, se dice que el código no es vectorizable porque importa el orden de ejecución y un cambio podría afectar el resultado del programa. En los ejemplos 1 y 2 se pueden observar dos ejemplos distintos de casos no vectorizables.

En el caso del código 1 tenemos una dependencia *RAW* (*Read After Write*) en la cual hay una lectura en una iteración que depende de la escritura en una iteración anterior, mientras que en el caso del código 2 (*Write After Write*) hay una escritura en una misma posición de memoria (en este caso la variable `acum`) en iteraciones sucesivas.

Cache

La memoria cache es un tipo de memoria integrada en el chip de uno o más procesadores, que tiene por objetivo incrementar la performance del sistema al almacenar datos de uso recurrente o próximo. Esta memoria cache es mucho más rápida que la memoria principal, aunque como desventaja tiene que es de mucha menor capacidad y mucho más cara en términos económicos. Su surgimiento responde a problemas de cuello de botella provocados por la memoria principal a mediados de los 80 [18].

La figura 1.4 muestra la evolución en términos de desempeño del procesador y la memoria. Queda claro que ambos dispositivos muestran una clara diferencia en su evolución a partir del a no 1985 como hemos mencionado y que, en estas circunstancias, el acceso a la memoria se ha convertido en el principal cuello de botella en términos de rendimiento para el sistema.

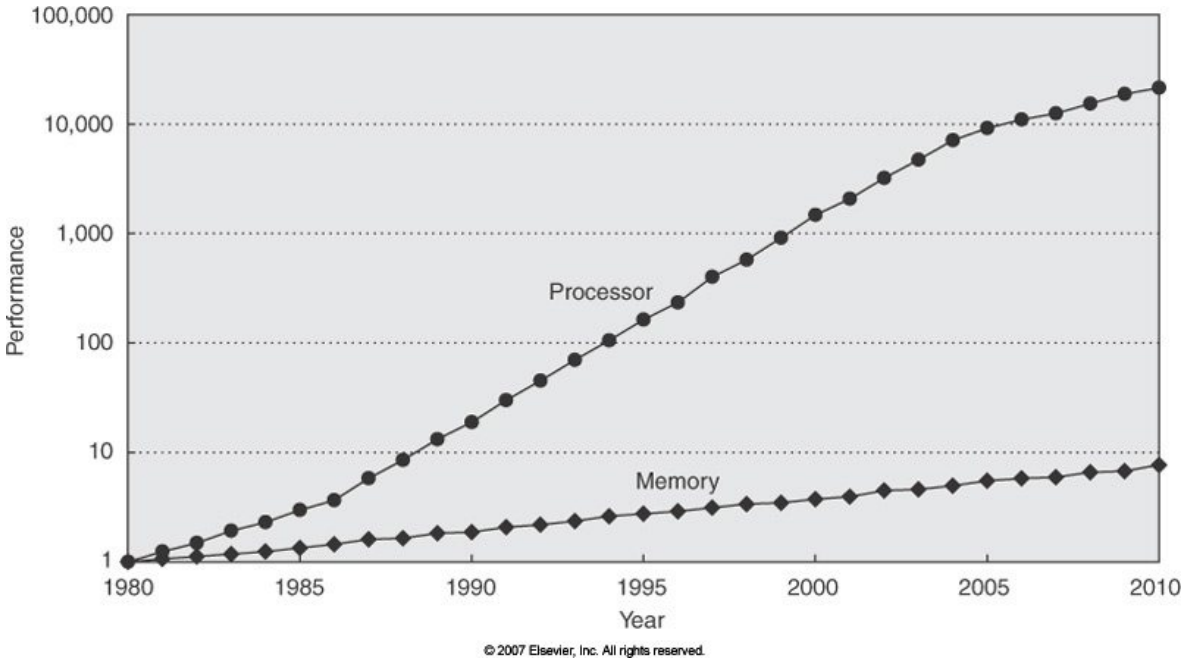

Fig. 1.4: Evolución de la performance dada por cpu y por memoria tomando como base el a no 1980.

Para que un programa haga un uso efectivo de la memoria cache, éste debe contar con *localidad espacial* y *localidad temporal*. La localidad espacial se basa en que los programas suelen procesar los elementos que se encuentran próximos entre sí, de este modo el procesador al guardarlos en cache, puede accederlos de manera muy eficiente. La localidad temporal refiere a que durante la ejecución de un programa, el uso de un dato o conjunto de datos suele ocurrir múltiples veces en un intervalo corto de tiempo, lo que implica que al guardarlos en cache, las sucesivas lecturas serán mucho más rápidas.

Multithreading

El multithreading tiene sus orígenes en la década de los 60, aunque las implementaciones más similares a las utilizadas hoy en día datan de la década de los 80 [19]. Es un modelo de programación y ejecución de instrucciones que permite que múltiples hilos o *threads* compartan recursos dentro de un mismo proceso pero siendo ejecutados independientemente. Esto abre la posibilidad de realizar tareas en forma paralela y permite mejoras de performance siempre y cuando las tareas a ejecutar sean independientes entre sí. Un ejemplo simple y real puede observarse en la figura 1.5: como cada elemento de la matriz que resulta de realizar la multiplicación es calculado mediante un único par fila-columna, el trabajo puede ser repartido al asignar una fila de la matriz izquierda a cada thread, llevando el `thread1` la primera fila, el `thread2` la segunda fila y así.

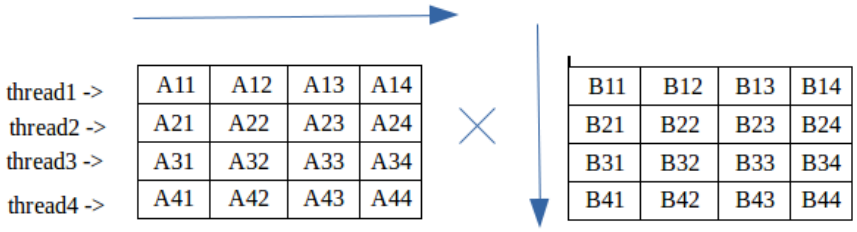

Fig. 1.5: Diagrama de cómo se podría repartir el trabajo entre distintos *threads* al realizar una multiplicación de dos matrices.

Procesadores multicore

Los primeros procesadores de consumo masivo con varios núcleos fueron lanzados por AMD en el año 2005 y luego por Intel en 2006, aunque para usos científicos, industriales y militares existen desde hace décadas [14]. Estos procesadores *multicore* son, básicamente, varios procesadores en un mismo chip, aunque dependiendo del modelo algunos comparten uno o más niveles de memoria cache.

La importancia de poseer soporte en hardware para realizar múltiples tareas al mismo tiempo ser una muy buena solución para continuar con el incremento en el desempeño de las aplicaciones, ya que por limitaciones tecnológicas incrementar la velocidad de reloj de los procesadores era inviable (provocaba graves problemas de consumo y de temperatura). Este soporte de hardware, sin embargo, depende exclusivamente de que el software sea diseñado y programado exponiendo paralelismo, con lo cual se hace imperativo un buen aprovechamiento de recursos y buenos programadores. Un ejemplo de la arquitectura de múltiples procesadores puede verse en la figura 1.6 que presenta una visión de la arquitectura de un sistema con dos procesadores. Cada procesador tiene, a su vez, dos núcleos. Los sistemas actuales no tienen una única memoria cache, sino que se separa en niveles (cuanto más alto el nivel de la cache, más lento y más grande). Los núcleos tienen una cache propia y pequeña L1, mientras que la cache L2 es compartida. En este ejemplo, solo hay dos niveles de cache, pero en los procesadores actuales, que poseen más núcleos, también hay una cache L3 compartida por varios núcleos, inclusive podría ser accedida por todos.

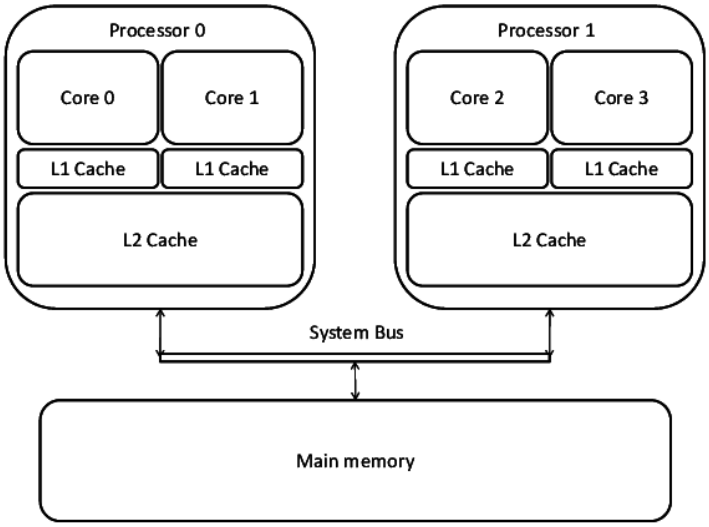

Fig. 1.6: Diagrama arquitectural simple de un sistema que posee dos procesadores con dos núcleos cada uno. Puede observarse que cada núcleo tiene una caché privada, mientras que dentro de cada procesador comparten una cache también. Existen otras arquitecturas de cache, como por ejemplo que cada núcleo posea caches privadas L1 y L2, y que compartan una cache más grande y más lenta llamada L3.

Procesamiento con GPU

Las GPU (Graphic Processing Units) son hardware especializado que usualmente eran utilizadas únicamente en el ámbito de los videojuegos y la edición de video, por sus capacidades de realizar cómputos simples en cantidades enormes de datos en muy poco tiempo, algo que tradicionalmente las CPU no eran capaces de hacer (básicamente porque hay un *tradeoff* entre poder realizar pocas operaciones complejas versus poder realizar muchas operaciones simples). Un ejemplo de un cómputo simple podría ser multiplicar una gran cantidad de elementos de un arreglo por una constante.

Con el advenimiento de ciertas API/estándares como CUDA [20] y OpenCL [21] junto con las mejoras en las capacidades de cómputo de GPUs, su uso para procesamiento masivo de datos se extendió a aplicaciones científicas e industriales. Hoy en día, técnicas de inteligencia artificial tienen el uso de las GPU como su principal dispositivo de cómputo (por ejemplo *Deep Learning* [22]). En esta tesis no se presentarán resultados para este tipo de hardware, pero se estima que los aportes realizados permitirán avanzar en este camino en un futuro.

En la figura 1.7 se puede observar una arquitectura simplificada de una GPU moderna. A simple vista se puede observar una gran diferencia respecto de una arquitectura de CPU como la de la figura 1.6. Los procesadores suelen tener una pequeña cantidad de núcleos con un alto poder de procesamiento y una poderosa máquina de soporte para reordenamiento de instrucciones en vuelo, predicción de saltos, etc. Las GPUs se basan en un diseño que hace énfasis en la simplicidad y masividad: los núcleos se enfocan en la realización de una gran cantidad operaciones aritméticas, pero para poder hacerlo, el programa debe estar diseñado para exponer este tipo de paralelismo. Para usar las placas de video, se necesario programar de nuevo la aplicación teniendo en cuentas las características de la plataforma.

Es lo más común que al intentar utilizar ana aplicación programada para procesadores tradicionales en GPU, su performance sea muy mala. Nuevamente, se necesitan buenos programadores para poder obtener los potenciales beneficios de esta tecnología.

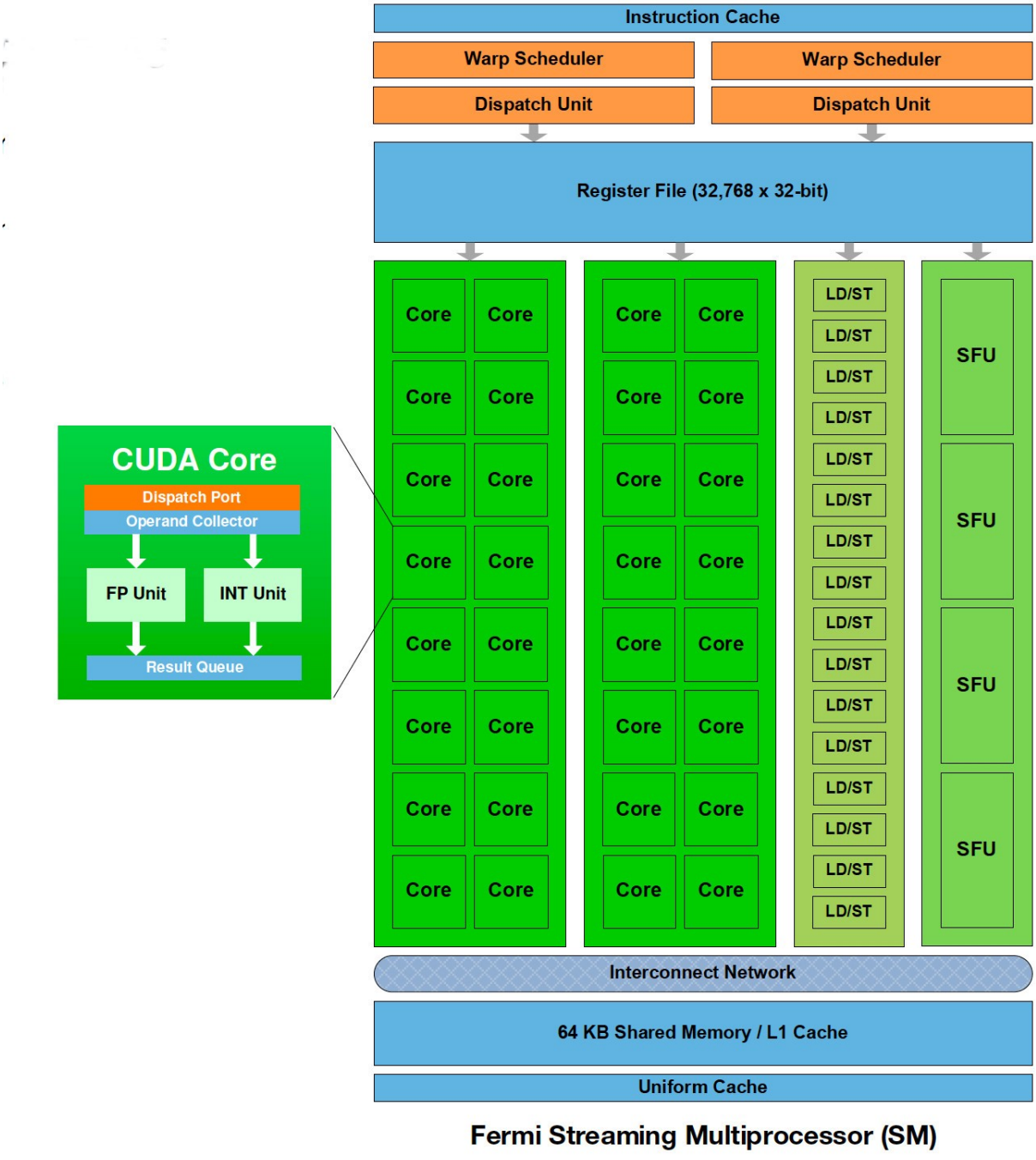

Fig. 1.7: Diagrama arquitectural simple de una GPU moderna de NVIDIA. Los núcleos descriptos son capaces de operaciones matemáticas menos complejas, pero dada su gran cantidad (por ejemplo, una `RTX 2080 Ti` posee más de 4000 núcleos CUDA) el *throughput* de operaciones vectoriales es ciertamente muy alto.

En este capítulo hemos introducido los aspectos fundamentales de la teoría molecular, cuya implementación computacional proponemos optimizar, así como también hemos introducido los conceptos relevantes de computación con el fin de facilitar el entendimiento de lo expuesto en los captítulos siguientes.

2. NANOPORE

En este capítulo haremos un análisis detallado de los aspectos más importantes del programa, así como su entorno de ejecución, bibliotecas de cálculo científico, compiladores y hardware utilizado. Además, se realiza un análisis pormenorizado de los distintos aspectos que afectan el desempeño de la resolución numérica de la teoría molecular.

2.1. Implementación inicial del programa

El programa, al que llamaremos de aquí en más **Nanopore**, fue desarrollado utilizando `FORTRAN`. `FORTRAN` (cuyo nombre se deriva de **FOR**mula **TRAN**slator) es un lenguaje de programación originalmente desarrollado por IBM en los años 50 cuyo foco fue el de las aplicaciones numéricas. A lo largo de las décadas, fue evolucionando por medio de la aparición de distintos estándares y sigue siendo utilizado por una gran cantidad de programadores en todo el mundo (sobre todo en el ámbito académico de ciencias fisicoquímicas y físicas, ya que es muy conveniente para operaciones de álgebra matricial y hay una gran cantidad de utilidades desarrolladas por la comunidad). Hoy en día, la versión de lenguaje aceptado por la comunidad es el **Fortran 90**.

Adicionalmente, en el contexto de la aplicación que se está analizando en este trabajo, se utilizan dos bibliotecas principales:

- `OpenMPI` [23]: Es una implementación de código abierto de MPI (*Message Passing Interface*). Está orientada a realizar la distribución y sincronización de un trabajo en distintos colaboradores que pueden ser tanto procesos ejecutando en un mismo nodo (es decir, distintos núcleos del mismo servidor bajo una arquitectura de memoria compartida) como procesos ejecutando en nodos interconectados por una red siguiendo una arquitectura tipo cluster (memoria distribuida).
- `Sundials Kinsol` [24]: Es una biblioteca que provee facilidades para la resolución numérica de sistemas de ecuaciones algebraicas no lineales.

2.1.1. Entorno de ejecución

La tabla 2.1 incluye una lista de los componentes utilizados para la ejecución de las distintas pruebas iniciales. Para las pruebas posteriores, que involucran cambios en los componentes de software o de hardware, se detallarán las modificaciones respecto a esta configuración original.

A continuación hacemos una descripción general de la arquitectura del programa.

2.1.2. Arquitectura de Nanopore (serial)

En la figura 2.1 se introduce la arquitectura de Nanopore cuando es ejecutado en un único nodo. La misma comprende los siguientes componentes:

- **Definitions**: En este módulo se procede a leer los parámetros de entrada de Nanopore, que se encuentran almacenados en un archivo de configuración llamado `DEFINITIONS.txt`

Sistema operativo	`Ubuntu 14.04`, kernel `3.13.0-32-generic`
MPI	`OpenMPI 1.8.7`
Solver	`Sundials Kinsol 2.8.1`
Compilador	`GNU Fortran compiler 4.8.4`
Flags del compilador	`-O3`
CPU	Intel(R) Core(TM) `i5-3330 CPU` 3.00 GHz, 4 núcleos
RAM	16 GB, 2×Kingston DDR3 8 GB 1333 MHz
GPU	`Tesla C2070`, drivers `352.07`
HDD	`WD Blue` 1 TB Desktop Hard Disk Drive - 7200 rpm SATA 6 Gbit/s 64 MB Cache

Tab. 2.1: Entorno de ejecución que será utilizado en los experimentos numéricos de esta tesis, salvo que se especifique lo contrario.

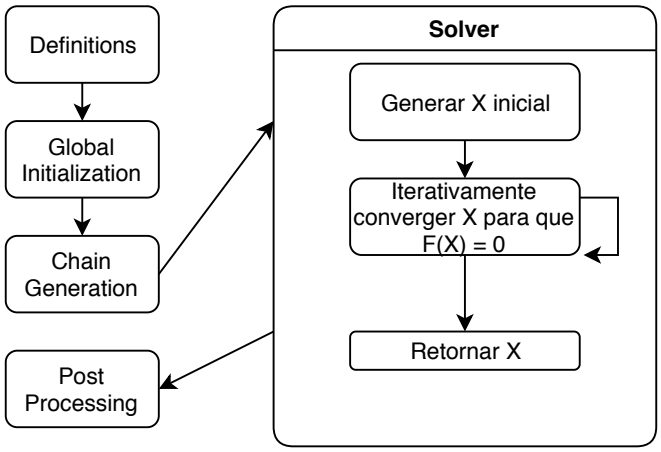

Fig. 2.1: Arquitectura general de Nanopore para el caso de ejecución en serie

- **Global Initialization**: En este módulo se inicializan las variables necesarias y se realizan los pedidos de memoria al sistema operativo para luego poder guardar las estructuras principales que hacen a la simulación del sistema bajo estudio. Según algunos parámetros de entrada definidos anteriormente, las inicializaciones pueden ser a partir de archivos auxiliares o realizarse en tiempo de ejecución.
- **Chain Generation**: En este módulo, a partir de las variables e inicializaciones del modelo, se genera un conjunto aleatorio y representativo de conformaciones posibles para cada polímero presente en el sistema discretizado. El sistema, en el cual estos polímeros se encuentran, es un sistema de dos dimensiones con simetría axial $\langle r, z \rangle$, en el cual se discretizan las coordenadas $\langle r, z \rangle$. Cada coordenada del sistema tiene un porcentaje de solvente y un porcentaje de polímero, sumando ambos porcentajes el 100 % del contenido. Cada polímero es una cadena lineal de segmentos y la distancia y el ángulo que forman tres segmentos consecutivos son parámetros fijos en el modelo y definen una conformación posible.
- **Solver**: En este módulo podemos identificar varios pasos, mediante los cuales se intentará encontrar un vector $\mathbf{x}$ tal que $\mathbf{F}(\mathbf{x}) = 0$ (siendo $\mathbf{F}(\mathbf{x})$ un sistema de ecuaciones no lineales con funciones acopladas).

Más específicamente, el conjunto de ecuaciones $\mathbf{F}$ es la ecuación de empaquetamiento (ecuación 1.4 o, mejor dicho, su versión discretizada, la ecuación 1.9) igualada a cero. En esta ecuación se han reemplazado las expresiones provenientes de minimizar el funcional de energía libre (ecuaciones 1.7 y 1.8). La única incógnita remanente en esta ecuación es la fracción de volumen del solvente en cada celda, la cual constituye el vector $\mathbf{x}$. De esta forma, encontrar la solución $\mathbf{F}(\mathbf{x}) = 0$ implica encontrar la fracción de volumen del solvente en cada punto que minimiza la energía libre del sistema. A partir de esta variable, es posible calcular otras propiedades estructurales (por ejemplo, la densidad del polímero) y termodinámicas (por ejemplo, el valor de la energía libre).

El vector $\mathbf{x}$ representa la densidad de solvente en cada punto del sistema (cada posición z, r en la grilla). Tal como se vio en el capítulo 1, resolver la teoría molecular involucra encontrar el vector $\mathbf{x}$ que minimice la energía libre del sistema.

- Primero se genera el vector $\mathbf{x}$ inicial. Según la configuración del programa, este vector inicial puede ser $\mathbf{x} = 0$ o construido a partir de un archivo.
- Se provee a la biblioteca **Kinsol** de $\mathbf{x}$ y de una función creada por el usuario llamada `fkfun`, que es la encargada de calcular $\mathbf{F}(\mathbf{x})$.
- Luego de varias iteraciones utilizando la función `fkfun` se llega a un vector $\mathbf{x}$ tal que $\mathbf{F}(\mathbf{x}) = 0$. La biblioteca Kinsol utiliza el método de **Newton–Krylov** para resolver el sistema de ecuaciones no lineales. Esto se debe a que dicho método no necesita calcular Jacobianos, los cuales para este sistema serían prohibitivos de guardar en memoria.

- **Post Processing**: En este módulo se utiliza el ya calculado vector $\mathbf{x}$ para, entre otras cosas, obtener el peso estadístico de cada conformación del sistema. También se escriben en disco resultados útiles para visualización, como por ejemplo archivos *.vtk* que pueden ser leídos con *The Visualization Toolkit (VTK).*

2.1.3. Arquitectura de Nanopore (múltiples nodos)

- **Definitions**: En este módulo se procede a leer los parámetros de entrada de Nanopore, que se encuentran guardados en un archivo de configuración llamado *DEFINITIONS.txt*

- **Global Initialization**: No cambia respecto a lo anteriormente descripto, salvo aquellas inicializaciones propias de MPI.

- **Chain Generation**: En este módulo, la diferencia de comportamiento radica en que para cada nodo se genera un subconjunto del conjunto representativo de conformaciones. Esta generación se hace partiendo de una semilla aleatoria independiente para cada nodo.

- **Solver**: En este módulo, el comportamiento es más complejo que en la versión serial. El jefe (*master*) y cada uno de los nodos esclavos (*slaves*) trabajarán en paralelo utilizando cada uno un subconjunto del conjunto de conformaciones generado anteriormente. Puede verse una graficación de su comportamiento en la figura 2.2

  - El nodo jefe llama a Kinsol que, a partir del vector $\mathbf{x}_{\text{inicial}}$, genera un vector $\mathbf{x}_1$.

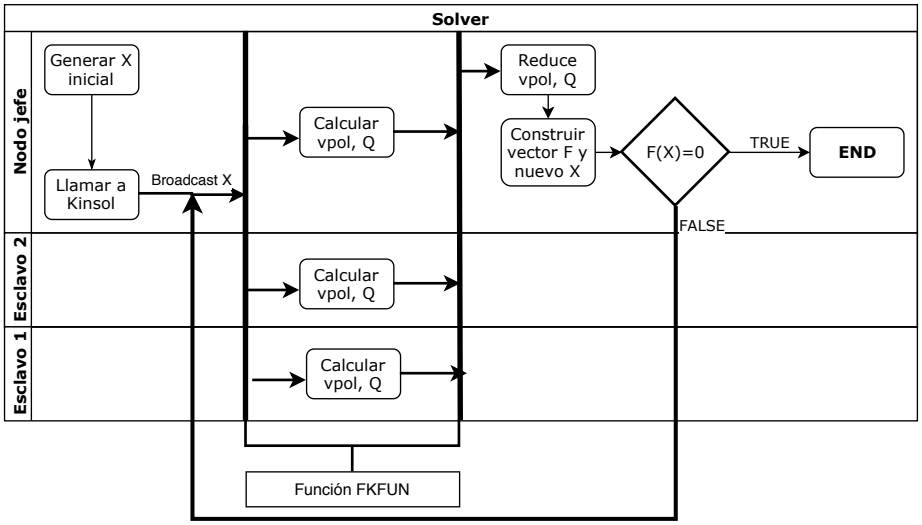

Fig. 2.2: Comportamiento del solver cuando se ejecuta en varios nodos utilizando MPI

- Kinsol llama a `fkfun` pasándole el vector $\mathbf{x}_1$.
- Dentro de `fkfun`, el nodo jefe envía mediante *broadcast* el vector $\mathbf{x}_1$ y un flag para indicar que hay trabajo por hacer a todos los nodos esclavos.
- Todos los nodos realizan cálculos cuyos resultados combinarán mediante un *reduce*. Cada nodo utiliza el mismo vector $\mathbf{x}_1$, pero un distinto subconjunto de conformaciones. Al terminar el cálculo, quedan a la espera de que el nodo *master* les indique si es necesaria otra iteración o si el trabajo terminó. En este momento el *master* calcula un nuevo $\mathbf{F}(\mathbf{x})$ (la energía libre).
- El nodo *master*, luego de haber calculado $\mathbf{F}(\mathbf{x_1})$, envía mediante *broadcast* a los nodos esclavos el resultado. Si $|\mathbf{F}(\mathbf{x_1})| < 1 \times 10^{-10}$, el trabajo termina (se envía flag = 0). De lo contrario, se vuelve a ejecutar `fkfun` pero con el vector de entrada $\mathbf{x}_2$ (modificado por kinsol a partir de $\mathbf{x}_1$), siendo ésta una nueva iteración donde trabajarán como se describió anteriormente.

- **Post Processing**: Su comportamiento no cambia respecto a lo anteriormente descripto.

Habiendo descripto el comportamiento con múltiples nodos, cabe aclarar que los análisis posteriores no involucrarán uso alguno de MPI y que la sección crítica en cuanto a performance no depende en sí misma de MPI. Dicha sección crítica es el módulo *fkfun* y será analizada a continuación.

2.2. Análisis de Nanopore

El objetivo principal del trabajo del cual deriva esta tesis fue identificar posibles problemas de performance en Nanopore, para lo cual fue necesario realizar un análisis (profiling) para llegar a las causas. El profiling es una herramienta utilizada para analizar de varias maneras el comportamiento de un programa, como por ejemplo el uso de memoria, el uso de procesador, el porcentaje de tiempo pasado en una sección del programa, etc. En el

caso particular de este proyecto, ya contábamos con información previa sobre la sección de código que más tiempo insumía, pero igualmente decidimos realizar el profiling para tener conclusiones objetivas y medibles.

2.2.1. Sobre los parámetros

El programa Nanopore cuenta con muchos parámetros configurables que son leídos desde un archivo de configuración. Para entender un programa de manera exitosa se lo puede pensar como una función matemática, la cual produce un cierto output dependiendo de sus inputs. Si tomamos el tiempo de ejecución, el uso de memoria RAM o el uso de cpu promedio como outputs, podemos fijar todos los parámetros y dejar uno libre para tenerlo como input variable y así poder sacar conclusiones que de otra manera serían difíciles o imposibles de obtener.

En general, para el análisis de Nanopore, se tendrá como variable libre la *cantidad de conformaciones por cadena*, que en el código fuente se la identifica por el nombre **cuantas**. En esta tesis, nos referiremos a esta variable como la cantidad de conformaciones, para que el lector pueda asociar más fácilmente las componentes computacionales y de teoría molecular que se mencionan.

2.2.2. Resultados de un análisis preliminar de Nanopore

En la figura 2.3 mostramos la variación en el tiempo de ejecución de Nanopore a medida que se incrementa la cantidad de conformaciones. A simple vista se puede ver que el tiempo total de ejecución del programa no crece linealmente respecto a la cantidad de conformaciones, aunque se puede decir que los resultados siguen una tendencia monótona creciente a medida que aumenta la cantidad de conformaciones.

La explicación de este fenómeno es que la cantidad de ejecuciones de `fkfun` no depende directamente de la cantidad de conformaciones, sino que depende de la forma que tenga la función $\mathbf{F}(\mathbf{x})$, como en todo proceso donde se quiere aproximar iterativamente el mínimo de un sistema de ecuaciones. Es decir, podría ocurrir que a pesar de que un mayor número de conformaciones haga más cara la ejecución de `fkfun`, la forma de F resulte en menos iteraciones para llegar a la solución.

En la figura 2.4 mostramos la variación en el uso de memoria de Nanopore a medida que se aumenta la cantidad de conformaciones[1]. Naturalmente se puede observar un uso creciente, principalmente dado por los pedidos de memoria necesarios para guardar el conjunto representativo de las conformaciones posibles.

Adicionalmente, cabe destacar que para problemas en los que la cantidad de conformaciones es tal que la memoria de un nodo no es suficiente, se puede ejecutar el programa en varios nodos, repartiendo la cantidad de conformaciones proporcionalmente en cada uno de ellos.

2.2.3. Análisis de `fkfun`

La función `fkfun` es la subrutina que se provee a la biblioteca Kinsol para que dado un $\mathbf{x}$ y una función F evalúe $\mathbf{F}(\mathbf{x})$, siendo esto lo necesario para luego de varias iteraciones

[1] Cuando mencionamos el *uso de memoria*, nos referimos al pico de uso de memoria residente, que es la porción de memoria RAM asignada a un programa.

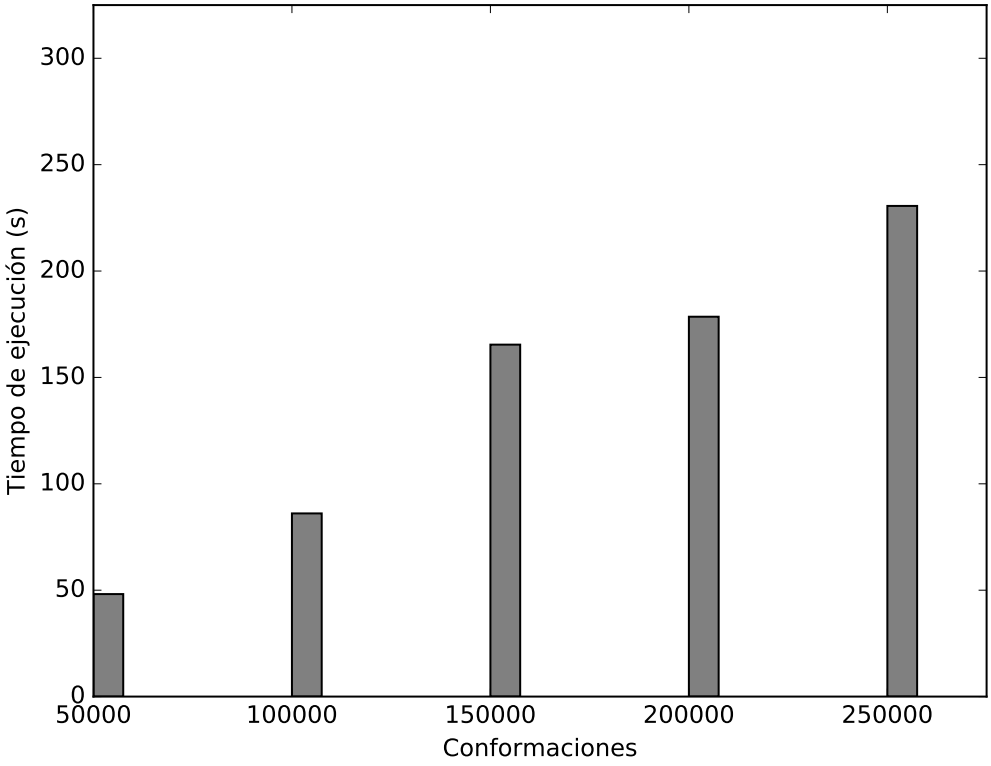

Fig. 2.3: Tiempos de ejecución del programa original a medida que se incrementa la cantidad de conformaciones.

encontrar un $\mathbf{x_n}$ tal que $\mathbf{F}(\mathbf{x_n}) = 0$. Esta subrutina posee varias etapas distintas para lograr su objetivo, las cuales son:

- Calcular **xpot**: El cálculo de este vector (que representa el peso estadístico que tendría un segmento de cadena en cada posición de la grilla del sistema, es decir, la expresión entre corchetes en la exponencial de la ecuación 1.8 para cada celda en el sistema) no presenta un gran costo de procesamiento en comparación al costo del código crítico.
- Calcular **avpol**: El cálculo de este vector (que representa la fraccion de volumen del polímero en cada posición de la grilla del sistema) representa el mayor costo y se lo considera el código crítico.
- Calcular la función **F**: El cálculo de esta función no presenta un gran costo de procesamiento en comparación al costo del código crítico.

Como ya mencionamos anteriormente, el módulo `fkfun` contiene la porción de código crítico que más tiempo consume del total del programa, por lo que es menester analizarlo en profundidad. El algoritmo 3 es una reproducción en pseudocódigo que captura el comportamiento de dicho código crítico (es importante aclarar que el código `FORTRAN` representado por el pseudocódigo no tiene una correspondencia exacta, aunque representa fielmente el comportamiento del mismo). Se puede observar que posee dos variables sobre

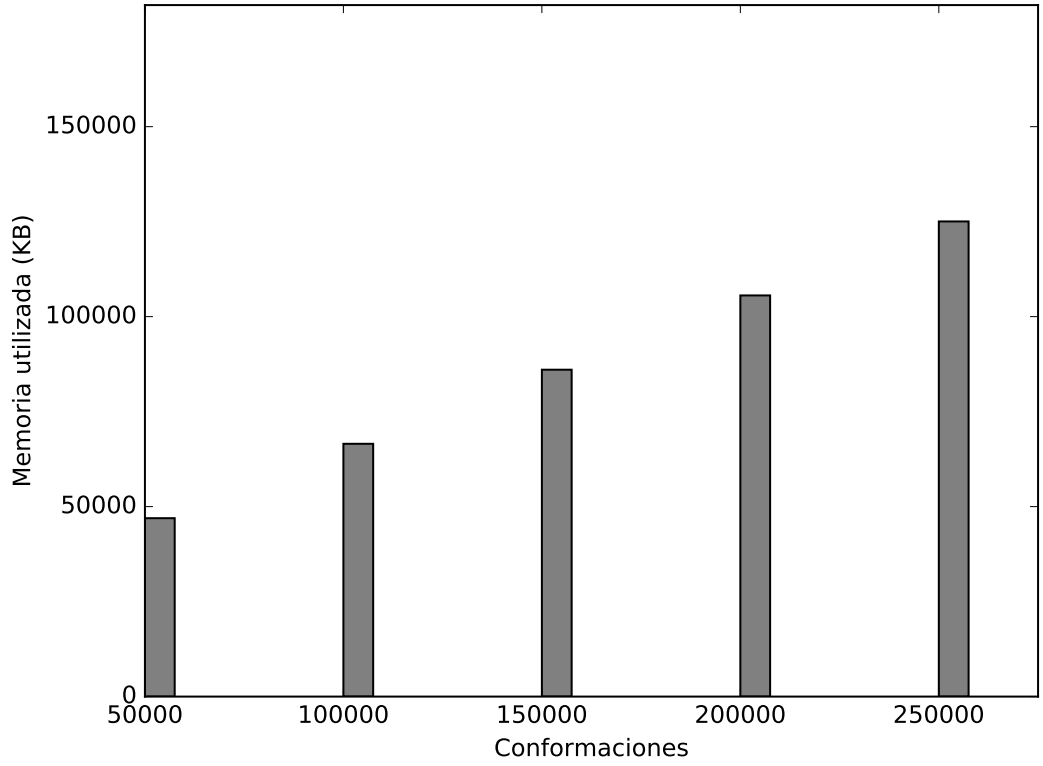

Fig. 2.4: Uso de memoria del programa original amedida que se incrementa la cantidad de conformaciones.

las que se hacen ciclos, siendo la cantidad de conformaciones el ciclo externo y la cantidad de segmentos de la cadena los ciclos internos.

Esto nos permite deducir que la complejidad del algoritmo es $\mathcal{O}(\text{conformaciones} * \text{segmentos})$, pero dado que anteriormente aclaramos que la única variable libre del programa sería la cantidad de conformaciones (y por lo tanto la cantidad de segmentos por cadena es fija) la complejidad del algoritmo es del orden de $\mathcal{O}(\text{conformaciones})$. Así mismo, dentro del mismo gráfico adjuntamos la recta tiempo/conformaciones que confirma experimentalmente dicha linealidad.

Dicho lo anterior, es importante mostrar y cuantificar experimentalmente lo mencionado, por lo que procederemos a describir algunos resultados experimentales que confirman lo dicho sobre el código crítico en cuanto a su costo relativo y su escalabilidad lineal.

En la figura 2.5 podemos observar que el tiempo de ejecución del código crítico escala linealmente a medida que se incrementa la cantidad de conformaciones posibles del sistema, confirmando que la complejidad es del orden de $\mathcal{O}(\text{conformaciones})$.

En la figura 2.6 podemos observar que el código crítico es la parte más costosa de todo el programa, y que esa relación de costo se mantiene a medida que se incrementa la cantidad de conformaciones.

Algorithm 3 Extracto del código que genera el mayor tiempo de ejecución en el programa original

$Q \leftarrow 0$
$avpol \leftarrow 0$
for $conformacion \leftarrow conformaciones\ de\ cadena$ **do**
  $lnpro \leftarrow 0$
  for $segmento \leftarrow segmento\ de\ cadena$ **do**
    $posicion \leftarrow grilla(conformacion, segmento)$
    $lnpro \leftarrow lnpro + xpot(posicion)$
  end for
  $pro \leftarrow e^{lnpro}$
  $Q \leftarrow Q + pro$
  for $segmento \leftarrow segmento\ de\ cadena$ **do**
    $posicion \leftarrow grilla(conformacion, segmento)$
    $avpol(posicion) \leftarrow avpol(posicion) + pro * factorproporcional$
  end for
end for

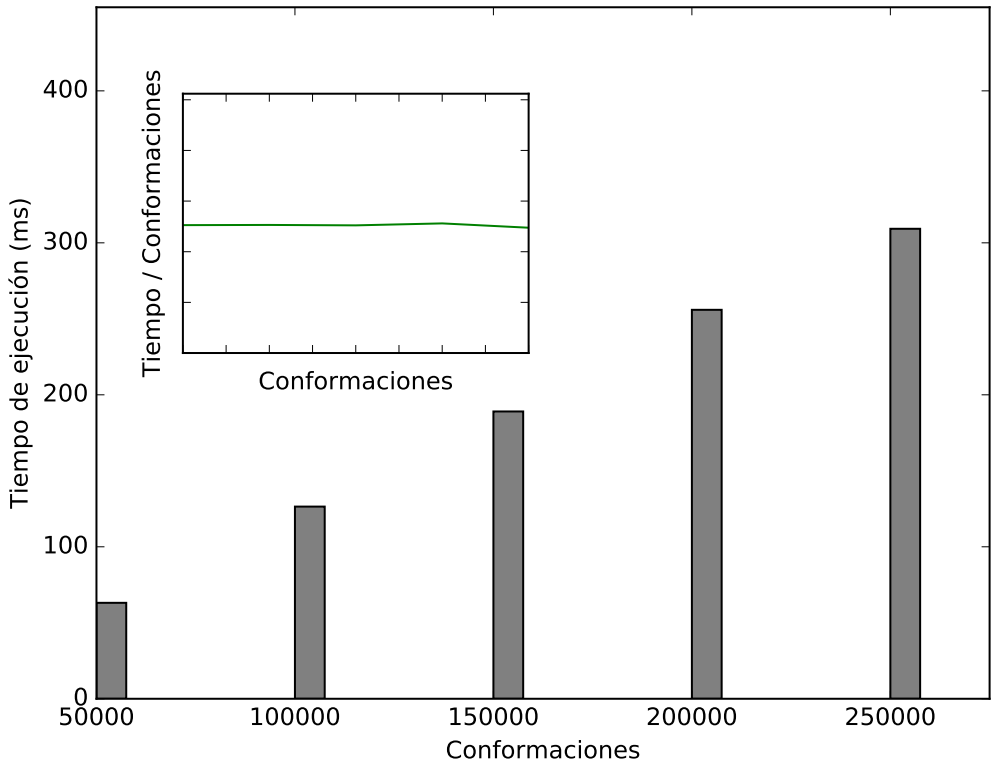

Fig. 2.5: Tiempo de ejecución del código crítico a medida que se incrementa la cantidad de conformaciones. Se incluye un gráfico interior que muestra la linealidad del incremento del tiempo de ejecución.

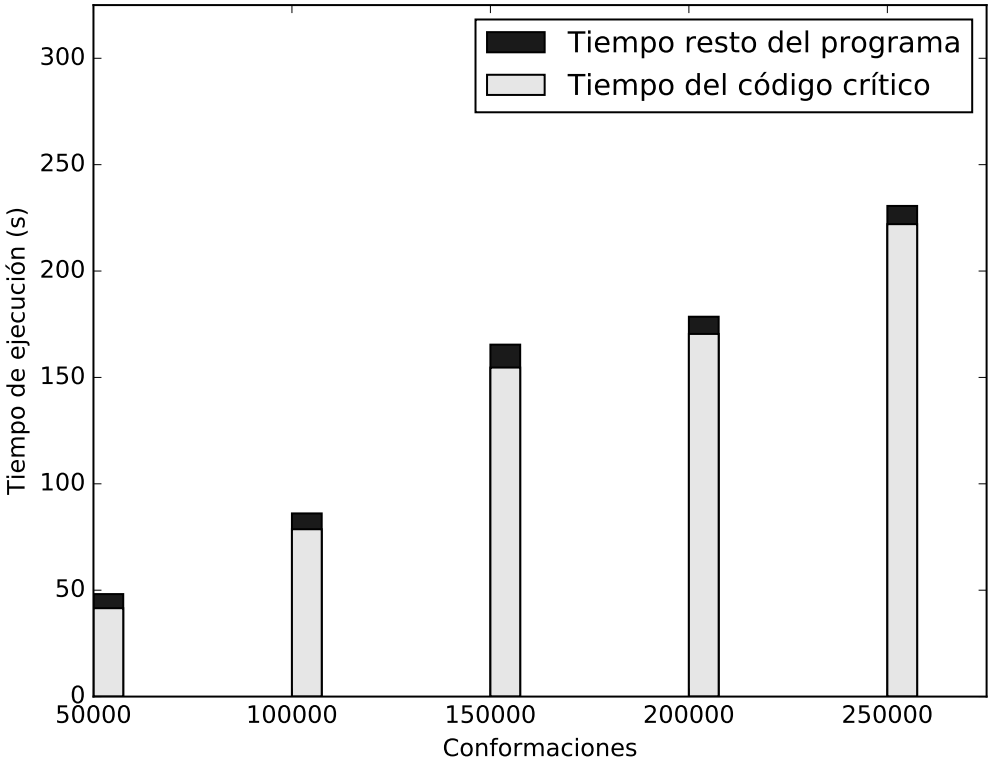

Fig. 2.6: Tiempo de ejecución del código crítico en relación al tiempo de ejecución total. Notar cómo se mantiene más o menos igual el tiempo que consume el resto del programa, mientras el tiempo del código crítico aumenta.

2.3. Análisis del código crítico

Habiendo cuantificado empíricamente la incidencia del código crítico en el tiempo de ejecución de Nanopore, corresponde pasar a hacer un análisis en detalle del comportamiento del algoritmo que se detalla en el algoritmo 3.

2.3.1. Patrones de acceso a memoria

Es esencial cuando guardamos datos en memoria de manera secuencial (ya sean arreglos o matrices) que el posterior acceso al mismo sea hecho de la manera adecuada para asegurar el máximo desempeño, aprovechando las propiedades de la localidad de referencias [25].

Lo primero que analizamos, al ser el programa hecho en FORTRAN, es si los arreglos de dos o más dimensiones son leídos en *column major order*. Esto implica, como podemos ver en la figura 2.7, que las columnas de la matriz se guardarán de manera contigua en memoria y que, por lo tanto, lo preferible es acceder a los datos de la matriz en ese mismo orden con el objetivo de aprovechar la velocidad de la memoria cache del procesador (recordemos, como se explica en el capítulo 1, que si el contenido de una posición de memoria es guardado en cache, es probable que los datos contiguos también lo estén por el fenómeno de localidad espacial).

A nivel código se logra esto si los índices que se usan yendo de izquierda a derecha se corresponden con los ciclos más anidados utilizados para recorrer la matriz. Fallar en seguir

estas convenciones resultará en *cache misses*, lo cual obviamente resulta en un programa más lento.

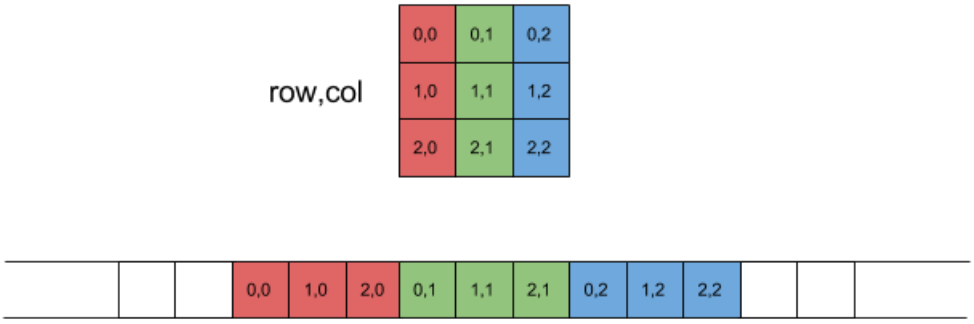

Fig. 2.7: Esquema de guardado en memoria de una matriz en Fortran, que muestra el caso de un arreglo de dos dimensiones cuyas columnas son guardadas de arriba hacia abajo, de izquierda a derecha.

Si observamos nuevamente el algoritmo 3, podremos notar que cuando se obtiene la posición de la grilla no se realiza del modo óptimo descrito anteriormente. Es decir, el ciclo externo debería recorrer los segmentos, y el interno las conformaciones.

Otro comportamiento observado en el código crítico es el de acceso a datos mediante indirecciones. Esto se ve reflejado cada vez que se obtiene la variable auxiliar `posicion`: No hay ninguna garantía de que haya localidad espacial entre las referencias de la variable `posicion` y la variable `xpot(posicion)`. Idéntica situación ocurre con la variable `avpol`.

2.3.2. Expresión matricial del código crítico

Considerando los patrones de memoria observados, presentamos a continuación un algoritmo equivalente al algoritmo 3 que no posee problemas de indirecciones en acceso a datos ni problemas con el recorrido de datos guardados en esquema de memoria por columna: la expresión matricial del código crítico. También demostraremos que dicho algoritmo produce los mismos resultados que su versión original.

Algorithm 4 Version matricial del código crítico

$lnpro \leftarrow numero * xpot$
$pro \leftarrow exp(lnpro)$
$Q \leftarrow \sum_{i=1}^{pos} \mathbf{pro}_i$
$avpol \leftarrow pro * numero * factorproporcional$

En el algoritmo 4 podemos observar la forma matricial del código crítico. En comparación a su versión anterior, surgen algunas diferencias y aclaraciones a notar:

- En lugar de `grilla`, ahora tenemos una matriz $\underset{(conformaciones \times pos)}{numero}$, siendo `pos` la cantidad de posiciones en la grilla del sistema y `conformaciones` la cantidad de conformaciones distintas.
- El vector `xpot` tiene tamaño `pos` y tiene el mismo contenido que la versión original.

- El vector `avpol` tiene tamaño `pos` y tiene el mismo contenido que la versión original.
- El vector `lnpro` tiene tamaño `conformaciones` y tiene el mismo contenido que la versión original.
- La función `exp` calcula el valor e^{x_i}, donde cada x_i es un elemento del vector que toma como parámetro de entrada.
- Los escalares `Q` y `factorproporcional` tienen el mismo contenido que la versión original.

Dicho esto, pasaremos a mostrar con un ejemplo que la versión matricial del código crítico produce los mismos valores `Q` y `avpol`.

Supongamos que el sistema cuenta con los siguientes valores `conformaciones`= 3, `pos`= 5 y `segmentos`= 2. En el código original, `lnpro` se calcula de la siguiente manera:

$$lnpro_i = \sum_{j=1}^{segmentos} \mathbf{xpot}(\mathbf{grilla}(i,j)) \tag{2.1}$$

donde $lnpro_i$ es cualquiera de las $i = (1..conformaciones)$ celdas de `lnpro` y `grilla` es una matriz de (conformaciones × segmentos) que almacena una posición $k \in (1..pos)$ del segmento j de la conformación i. En el código matricial, `lnpro` se calcula de la siguiente manera:

$$lnpro_i = \mathbf{numero_i} * \mathbf{xpot} = \sum_{k=1}^{pos} \mathbf{numero_i}(k) * \mathbf{xpot}(k) \tag{2.2}$$

donde $lnpro_i$ es cualquiera de las $i = 1..conformaciones$ celdas y $numero_i$ es la iésima fila de la matriz `numero`, que guarda la cantidad de segmentos para una combinación (k, i).

Para desarrollar las equivalencias, terminamos ahora definiendo el contenido de `grilla` y de `numero`:

$$\mathbf{grilla} = \begin{bmatrix} 1 & 1 \\ 3 & 2 \\ 1 & 3 \end{bmatrix} \mathbf{numero} = \begin{bmatrix} 2 & 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 & 0 \end{bmatrix} \mathbf{xpot} = \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \end{bmatrix}$$

Con las definiciones que necesitábamos ya construidas, mostramos que $lnpro_i$ es igual en ambas fórmulas:

$$lnpro_1 = \sum_{j=1}^{segmentos} \mathbf{xpot}(\mathbf{grilla}(1,j)) = \mathbf{xpot}(\mathbf{grilla}(1,1)) + \mathbf{xpot}(\mathbf{grilla}(1,2)) = 2 * x_1$$

$$= \mathbf{numero}_1(1) * \mathbf{xpot}(1) = \sum_{k=1}^{pos} \mathbf{numero}_1(k) * \mathbf{xpot}(k) = \mathbf{numero}_1 * \mathbf{xpot} \tag{2.3}$$

$$lnpro_2 = \sum_{j=1}^{segmentos} \mathbf{xpot}(\mathbf{grilla}(2,j)) = \mathbf{xpot}(\mathbf{grilla}(2,1)) + \mathbf{xpot}(\mathbf{grilla}(2,2)) = x_2 + x_3$$

$$= \mathbf{numero}_2(2) * \mathbf{xpot}(2) + \mathbf{numero}_2(3) * \mathbf{xpot}(3) = \sum_{k=1}^{pos} \mathbf{numero}_2(k) * \mathbf{xpot}(k)$$

$$= \mathbf{numero}_2 * \mathbf{xpot} \quad (2.4)$$

$$lnpro_3 = \sum_{j=1}^{segmentos} \mathbf{xpot}(\mathbf{grilla}(3,j)) = \mathbf{xpot}(\mathbf{grilla}(3,1)) + \mathbf{xpot}(\mathbf{grilla}(3,2)) = x_1 + x_3$$

$$= \mathbf{numero}_3(1) * \mathbf{xpot}(1) + \mathbf{numero}_3(3) * \mathbf{xpot}(3) = \sum_{k=1}^{pos} \mathbf{numero}_3(k) * \mathbf{xpot}(k)$$

$$= \mathbf{numero}_3 * \mathbf{xpot} \quad (2.5)$$

Habiendo mostrado las equivalencias de $lnpro_i$ en ambas versiones, la equivalencia de `Q` puede mostrarse de la siguiente manera:

$$Q_{original} = \sum_{i=1}^{conformaciones} pro_i = \sum_{i=1}^{conformaciones} e^{lnpro_i} = \sum_{i=1}^{conformaciones} pro_i = Q_{matricial} \quad (2.6)$$

Para finalizar, debemos mostrar la equivalencia de **avpol** en ambos algoritmos. Como podemos ver en el algoritmo 3, en **avpol** se acumulan distintas instancias de `pro` (multiplicadas por una constante) según qué pares conformación/segmento se considere. Esto significa que en **avpol**(k) se acumularán las instancias de `pro` que hayan sido creadas a partir de los pares conformación/segmento tales que **grilla**(conformación,segmento) = k. Teniendo esto en cuenta junto con los cálculos previos de $lnpro_i$, planteamos las siguientes equivalencias:

$$\mathbf{avpol_{original}}(1) = (2 * e^{lnpro_1} + e^{lnpro_3}) * fact = (2 * pro_1 + pro_3) * fact$$

$$= \mathbf{pro} * \mathbf{numero_1} * fact = \mathbf{avpol_{matricial}}(1) \quad (2.7)$$

$$\mathbf{avpol_{original}}(2) = e^{lnpro_2} * fact = pro_2 * fact = \mathbf{pro} * \mathbf{numero_2} * fact = \mathbf{avpol_{matricial}}(2) \quad (2.8)$$

$$\mathbf{avpol_{original}}(3) = (e^{lnpro_2} + e^{lnpro_3}) * fact = (pro_2 + pro_3) * fact$$

$$= \mathbf{pro} * \mathbf{numero_3} * fact = \mathbf{avpol_{matricial}}(3) \quad (2.9)$$

donde aquí $numero_j$ es la j-ésima columna de la matriz **numero**.

Mediante lo descrito al final de este capítulo, logramos encontrar una equivalencia para el código crítico expresada de manera matricial, lo que nos sirvió para implementar estructuras de datos para matrices esparzas. Esto nos permitió encontrar una solución superadora, que describiremos en el capítulo que sigue.

3. IMPLEMENTACIÓN

En este capítulo detallamos las diferentes alternativas que fueron consideradas para lograr una mejora en la performance del programa.

Para poder analizar exitosamente los cambios necesarios, se decidió trabajar con un programa nuevo que únicamente consistiera en la ejecución del código crítico una gran cantidad de veces, con la intención de minimizar la influencia del código de inicialización y obtener resultados reproducibles y no atribuibles a anomalías experimentales. Este nuevo programa base sobre el cual se trabajó será denominado **Nanopore Mock**.

3.1. Mejoras mediante optimizaciones de compilación

3.1.1. Compilador GNU Fortran

GNU Fortran es el compilador original utilizado en producción por **Nanopore**. En el Makefile de dicho proyecto, el único flag de optimización utilizado es *O3*, mientras que las bibliotecas linkeadas al programa relacionadas a **kinsol** son single thread (existiendo versiones multi thread). Estas decisiones seguramente están relacionadas a que **Nanopore** fue originalmente diseñado para utilizar MPI a lo largo de varios nodos, por lo cual no se tuvo en cuenta una versión multicore en un solo nodo.

El utilizar el flag **O3** es la decisión correcta, teniendo en cuenta que dentro de las optimizaciones simples que se pueden hacer, tiene una buena relación correctitud/velocidad. En pruebas preliminares, encontramos que el flag **Ofast** (que a las optimizaciones de O3 agrega el flag **fast-math** [26]) fue contraproducente, ya que la utilización del mismo no reportó mejoras de performance y provocó pequeñas diferencias numéricas.

No se realizaron mayores esfuerzos en encontrar una configuración superadora para este compilador, ya que por el análisis realizado en el capítulo 2, concluimos que la máxima mejora de rendimiento sería alcanzada mediante cambios en los algoritmos y estructuras de datos utilizados en el código crítico.

3.1.2. Compilador PGI de NVIDIA

Se realizaron pruebas con el objetivo de verificar la viabilidad de utilizar directivas de compilación de OpenACC [27], más específicamente las implementadas por el compilador PGI[1] de NVIDIA. Dichas directivas tuvieron por objetivo ejecutar el código en GPU, pero los resultados fueron desde ejecuciones muy lentas en comparación a la implementación original hasta directamente el no funcionamiento del programa, por lo que descartamos de lleno una implementación orientada a ejecutarse en GPU.

3.1.3. Mejoras mediante cambios en indexado

Como se mencionó en el capítulo anterior, los patrones de acceso a memoria del programa original no eran los óptimos a utilizar en una implementación en FORTRAN. Se procedió a migrar del patrón *row major* al patrón *column major*, pero no hubo cambios en cuanto a performance.

[1] `https://www.pgroup.com/index.htm`

3.2. Mejoras mediante cambios en estructuras de datos

Las estructuras de datos que describiremos a continuación son formas de almacenar matrices esparzas de manera tal que se optimiza la utilización de memoria por no guardar elementos iguales a cero. Su implementación fue llevada a cabo mediante la `Math Kernel Library` (de ahora en más abreviada **MKL**) de Intel en su versión con interfaz Fortran. Dicha biblioteca provee las estructuras de datos necesarias, además de funciones matriciales especialmente diseñadas para usarse con dichas estructuras. Cabe aclarar que la matriz esparza que nos interesa almacenar de mejor manera es la matriz *numero* del algoritmo 4, aunque aquí haremos referencia a una matriz genérica A.

3.2.1. Estructura CSR

La estructura CSR [28] (Compressed Sparse Row) almacena los elementos de la matriz (llamémosla A) de la siguiente manera:

- Un arreglo *values* que contiene únicamente los valores distintos a 0. Dicho arreglo es contruído recorriendo la matriz de izquierda a derecha y de arriba hacia abajo; este orden se denomina **row major order**. Es de tamaño N, donde N es la cantidad de elementos distintos a 0.
- Un arreglo *columns* que cumple la siguiente regla: El i-ésimo elemento de *columns* es el número de columna que contiene el i-ésimo elemento de *values*. Es de tamaño N.
- Un arreglo *pointerB* que cumple la siguiente regla: El j-ésimo elemento de *pointerB* contiene el índice del valor de *values* que sea el primer valor distinto a 0 en la fila j de la matriz A.
- Un arreglo *pointerE* que cumple la siguiente regla: El j-ésimo elemento de *pointerE* es el j+1-ésimo elemento de *pointerB*, y el último elemento de *pointerE* es el valor N+1.

Es decir, la matriz A:

$$A = \begin{bmatrix} 1 & -1 & * & 3 & * \\ -2 & 5 & * & * & * \\ * & * & 4 & 6 & 4 \\ -4 & * & 2 & 7 & * \\ * & 8 & * & * & -5 \end{bmatrix}$$

Se almacenará de la siguiente manera:

values = [1 , -1 , 3 , -2 , 5 , 4 , 6 , 4 , -4 , 2 , 7 , 8 , -5]
columns = [1 , 2 , 4 , 1 , 2 , 3 , 4 , 5 , 1 , 3 , 4 , 2 , 5]
pointerB = [1 , 4 , 6 , 9 , 12]
pointerE = [4 , 6 , 9 , 12 , 14]

Algorithm 5 Implementación de CSR

$mkl\_dcsrmv('N', conformaciones, pos, 1, desc, values, columns, pointerB, pointerE, xpot, 0, lnpro)$

$pro \leftarrow exp(lnpro)$
$Q \leftarrow \sum_{i=1}^{pos} \mathbf{pro}_i$
$mkl\_dcsrmv('T', conformaciones, pos, 1, desc, values, columns, pointerB, pointerE, pro, 0, avpol)$

$avpol \leftarrow avpol * factorproporcional$

En el algoritmo 5 podemos ver la implementación del código crítico usando la biblioteca **MKL**.

La función *mkl_dcsrmv* toma como parámetros los siguientes:

- Transponer o no la matriz. Si el valor es 'N', no se transpone. Si el valor es 'T',se transpone.
- La cantidad de filas de la matriz. En este caso, es la cantidad de conformaciones
- La cantidad de columnas de la matriz. En esta caso, es la cantidad de posiciones (pos).
- Un número por el cual se multiplicará la operación matriz-vector. En este caso es 1.
- Un arreglo de seis elementos que especifica propiedades de la matriz. En este caso, se configura que se indexa a partir de 1 (para usar el mismo indexado que Fortran) y que es una matriz general (es decir, no triangular, diagonal, etc).
- El arreglo **values** descrito anteriormente.
- El arreglo **columns** descrito anteriormente.
- El arreglo **pointerB** descrito anteriormente.
- El arreglo **pointerE** descrito anteriormente.
- El vector a multiplicar con la matriz. En este caso, xpot.
- Un número por el cual se multiplica al vector y se suma el resultado a la multiplicación matriz-vector. En este caso, 0.
- El vector destino donde se guardará el resultado de la multiplicación. En este caso, lnpro.

Entonces, volviendo al algoritmo 5, la línea 1 representa la siguiente fórmula de multiplicación matriz-vector:

$$lnpro = 1 * numero * xpot + 0 * xpot = numero * xpot$$

mientras que la línea 4 representa la siguiente fórmula:

$$avpol = 1 * numero^T * pro + 0 * xpot = numero^T * xpot$$

3.2.2. Estructura CSC

La estructura CSC (Compressed Sparse Column) almacena los elementos de la matriz (llamémosla A) de la siguiente manera:

- Un arreglo *values* que contiene únicamente los valores distintos a 0. Dicho arreglo es contruído recorriendo la matriz de arriba a abajo y de izquierda a derecha; Este orden es el mismo en el cual Fortran almacena matrices. Es de tamaño N, donde N es la cantidad de elementos distintos a 0.
- Un arreglo *rows* que cumple la siguiente regla: El i-ésimo elemento de *rows* es el número de fila que contiene el i-ésimo elemento de *values*. Es de tamaño N.
- Un arreglo *pointerB* que cumple la siguiente regla: El j-ésimo elemento de *pointerB* contiene el índice del valor de *values* que sea el primer valor distinto a 0 en la columna j de la matriz A.
- Un arreglo *pointerE* que cumple la siguiente regla: El j-ésimo elemento de *pointerE* es el j+1-ésimo elemento de *pointerB*, y el último elemento de *pointerE* es el valor N+1.

Es decir, la matriz A:

$$A = \begin{bmatrix} 1 & -1 & * & 3 & * \\ -2 & 5 & * & * & * \\ * & * & 4 & 6 & 4 \\ -4 & * & 2 & 7 & * \\ * & 8 & * & * & -5 \end{bmatrix}$$

Se almacenará de la siguiente manera:

values = [1 , -2 , -4 , -1 , 5 , 8 , 4 , 2 , 3 , 6 , 7 , 4 , -5]
rows = [1 , 2 , 4 , 1 , 2 , 5 , 3 , 4 , 1 , 3 , 4 , 3 , 5]
pointerB = [1 , 4 , 7 , 9 , 12]
pointerE = [4 , 7 , 9 , 12 , 14]

El algoritmo 6 muestra la implementación del código crítico usando la biblioteca **MKL**.

Algorithm 6 Implementación de CSC

$mkl\_dcscmv('N', conformaciones, pos, 1, desc, values, rows, pointerB, pointerE, xpot, 0, lnpro)$

$pro \leftarrow exp(lnpro)$
$Q \leftarrow \sum_{i=1}^{pos} \mathbf{pro}_i$
$mkl\_dcscmv('T', conformaciones, pos, 1, desc, values, rows, pointerB, pointerE, pro, 0, avpol)$

$avpol \leftarrow avpol * factorproporcional$

Como se puede ver, la función *mkl_dcscmv* es muy similar a la de CSR, salvo por usar **rows** en lugar de **columns**.

Con las implementaciones descritas en este capítulo tenemos las condiciones necesarias para realizar experimentos que diriman cuál es la mejor, y lo más importante, si son mejores que la implementación original. Es experimentos serán mostrados en el siguiente capítulo.

4. RESULTADOS

4.1. Resultados experimentales via mocks

Como explicamos en el capítulo anterior, se procedió a construir programas que únicamente contuvieran el código crítico de **Nanopore**. Adicionalmente, se procedió a exportar en varios archivos el contenido de la matriz esparza para diferentes configuraciones del programa original, para luego leerlos en cada experimento y lograr imitar el comportamiento original lo más precisamente posible.

Todos los resultados expuestos corresponden a ejecuciones de los programas en un único núcleo de procesador, **salvo que se especifique lo contrario**.

Las diferentes configuraciones surgen de variar la cantidad de conformaciones y la dimensionalidad del sistema; es decir, si el espacio en el cual las conformaciones se han proyectado es de una dimensión, dos o tres. A partir de estas configuraciones, se obtienen distintas matrices 2D cuya cantidad de columnas es el número de conformaciones, y su cantidad de filas es el número de posición dentro del sistema de proyecciones 1D, 2D o 3D (que son 50, 2500 y 12500 posiciones respectivamente).

Las figuras 4.1, 4.2 y 4.3 muestran la distribución espacial en la matriz de los elementos no nulos utilizando 25000 conformaciones, dependiendo de la dimensión. Puede observarse que los elementos no nulos se distribuyen en clusters. De manera similar, la cantidad de elementos nulos entre columnas también aumenta a medida que aumentan la dimensionalidad.

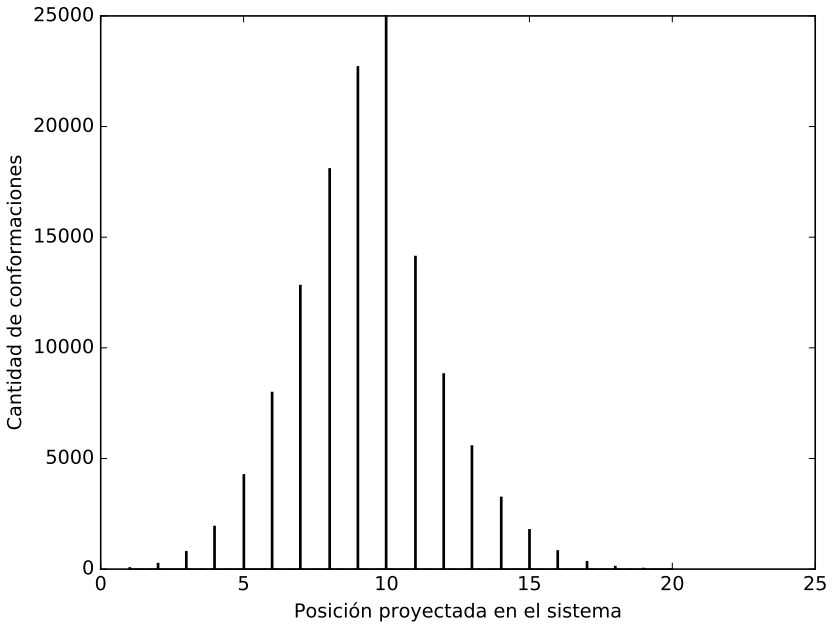

Fig. 4.1: Distribución de las conformaciones en dimensión 1D con densidad de 10.29216 % y cantidad promedio de columnas nulas entre clusters de conformaciones de 0

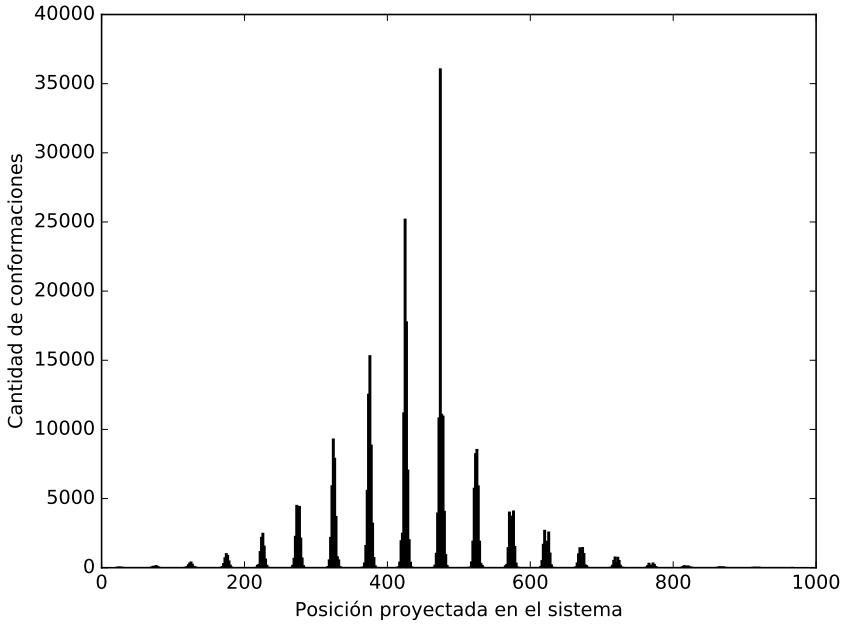

Fig. 4.2: Distribución de las conformaciones en dimensión 2D con densidad de 0.532104 % y cantidad promedio de columnas nulas entre clusters de conformaciones de 30

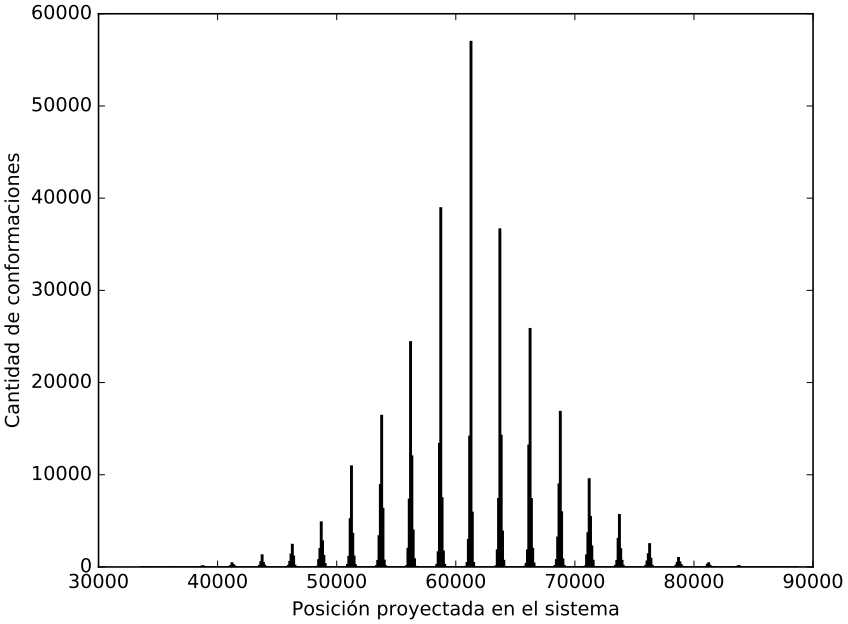

Fig. 4.3: Distribución de las conformaciones en dimensión 3D con densidad de 0.015145312 % y cantidad promedio de columnas nulas entre clusters de conformaciones de 115

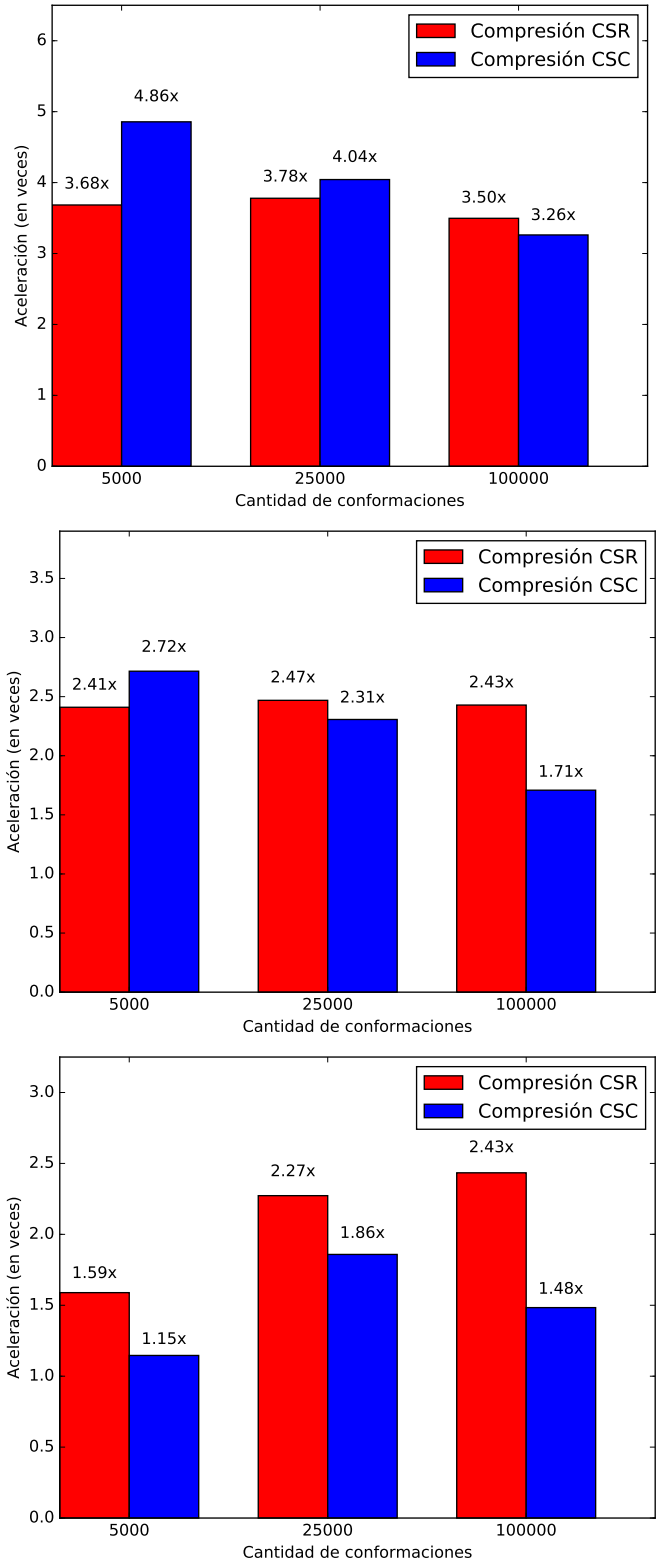

Fig. 4.4: Grado de mejora respecto a implementación original en matrices 1D, 2D y 3D respectivamente

La figura 4.4 muestra una comparación de las mejoras en tiempo de ejecución logradas por las estructuras respecto a la implementación original. Puede apreciarse como en 1D CSC tiene mejores resultados que CSR, en 2D llegan a estar relativamente parejos y en 3D CSR es el ganador en cualquier cantidad de conformaciones.

Una explicación a este resultado puede ser encontrada si miramos con atención los grados de mejora fijando el número de conformaciones en 25000 y variando la cantidad de posiciones proyectadas en el sistema, algo que expresamos en la figura 4.5.

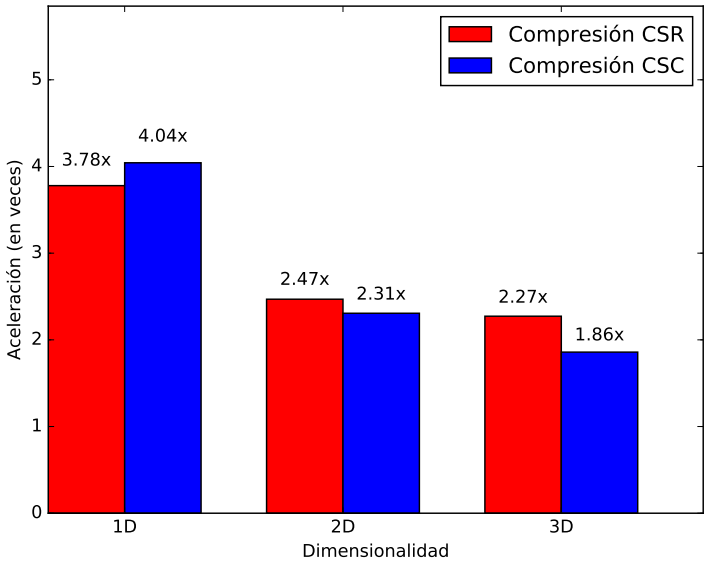

Fig. 4.5: Grado de mejora respecto a implementación original según dimensionalidad, con una cantidad fija de conformaciones

Si analizamos las distribuciones espaciales de las Figuras 4.1, 4.2, 4.3 y 4.5, podemos llegar a la conclusión de que a mayor cantidad de columnas nulas entre clusters, mejores resultados se logra usando la estructura CSR. Esto está estrechamente relacionado a que en CSR se indexan columnas, mientras que en la estructura CSC se indexan filas, lo que implica que habrá una mejor performance en distribuciones de datos que tengan más columnas nulas.

En cuanto al uso de memoria, podemos apreciar en la Tabla 4.1 que claramente estamos haciendo un trade-off sacrificando uso de memoria para obtener una mayor velocidad, que es el principal objetivo a lograr en este trabajo.

Estructura	**Cambio uso de memoria**	**Mejora de performance**
CSR	+116 %	+120 %
CSC	+70 %	+76 %

Tab. 4.1: Tradeoff memoria/performance respecto a la implementación original. 25000 conformaciones y dimensionalidad 3D

Una última comparación que queda realizar es el comportamiento del código crítico

cuando se distribuye el trabajo a realizar entre múltiples cores. La biblioteca MKL provee la capacidad de utilizar múltiples cores tan solo linkeando su versión multihilo y configurando las variables de entorno adecuadas.

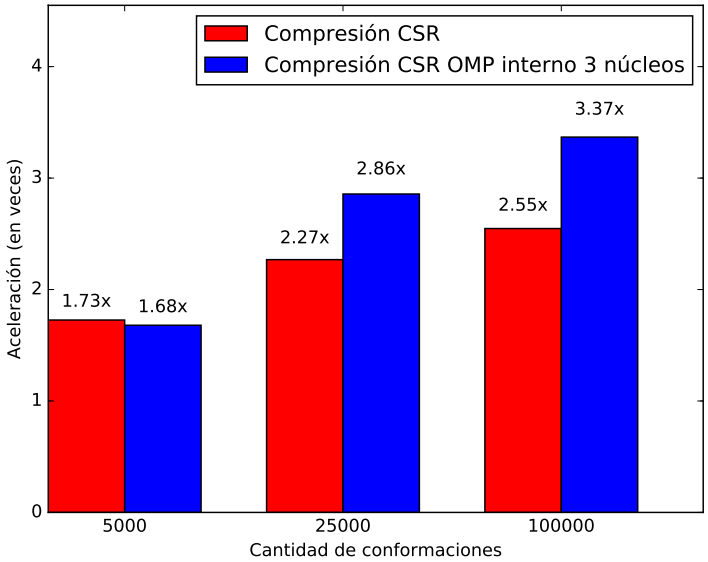

Fig. 4.6: Aceleración a medida que crece la cantidad de conformaciones, siendo la comparación entre la versión CSR del mock en un único core y CSR utilizando tres cores en su implementación interna de OpenMP

En la figura 4.6 podemos observar que la versión serial y la version utilizando tres núcleos de CSR incrementan la performance, pero que la versión de CSR paralelizada no obtiene buenos resultados.

Viendo que los resultados obtenidos fueron malos, decidimos implementar la paralelización del trabajo manualmente utilizando OpenMP. Dicha paralelización implicó lo siguiente, teniendo en cuenta que se realizó dividiendo a la matriz por filas equitativamente:

- Calcular la cantidad de trabajo que cada thread realizaría, repartiendo cierta cantidad de conformaciones a cada uno. Por ejemplo, repartir 1000 conformaciones resultaría en dos threads con 333 y uno con 334 conformaciones.
- Popular las estructuras que utiliza MKL para guardar la matriz *numero*, de manera al que cada thread acceda solamente a su parte de la matriz sin que ello implique multiplicar el uso de memoria.

Adicionalmente, ya vista la superioridad de CSR a CSC, decidimos únicamente implementar OpenMP en el algoritmo CSR.

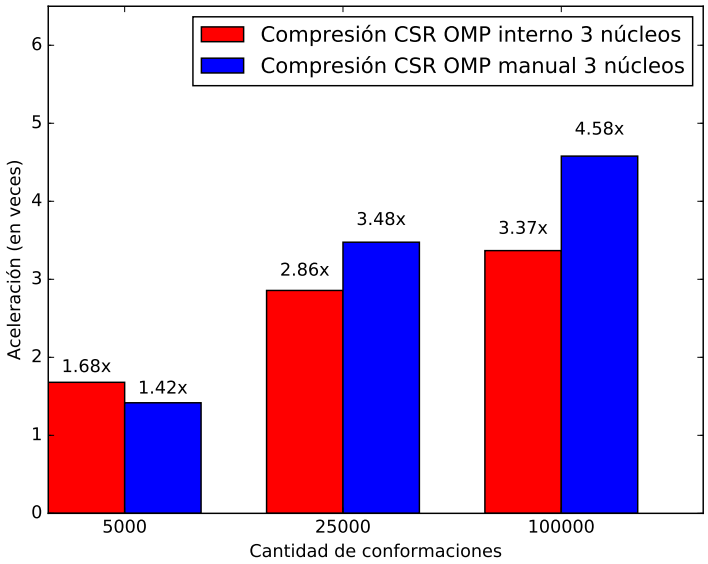

Fig. 4.7: Aceleración a medida que crece la cantidad de conformaciones, siendo la comparación entre la versión CSR utilizando tres núcleos en su implementación interna de OpenMP y CSR utilizando tres núcleo con implementación manual de OpenMP

Podemos apreciar en la figura 4.7 que efectivamente la implementación mediante OpenMP produce resultados mucho mejores.

4.2. Resultados experimentales en el programa original

Al resultar elegido CSR como la estructura de datos a implementar en el programa original, procedimos a evaluar su funcionamiento. Además de modificar el código crítico, fue necesario modificar el código de inicialización de las estructuras de datos pertinentes.

En la figura 4.8 podemos observar que la mejora con la implementación de CSR es evidente. También se observa una mejoría con respecto a los resultados obtenidos en la figura 4.4, lo que confirma nuestra hipótesis que el código mock de la implementación original no extrae perfectamente el comportamiento verdadero.

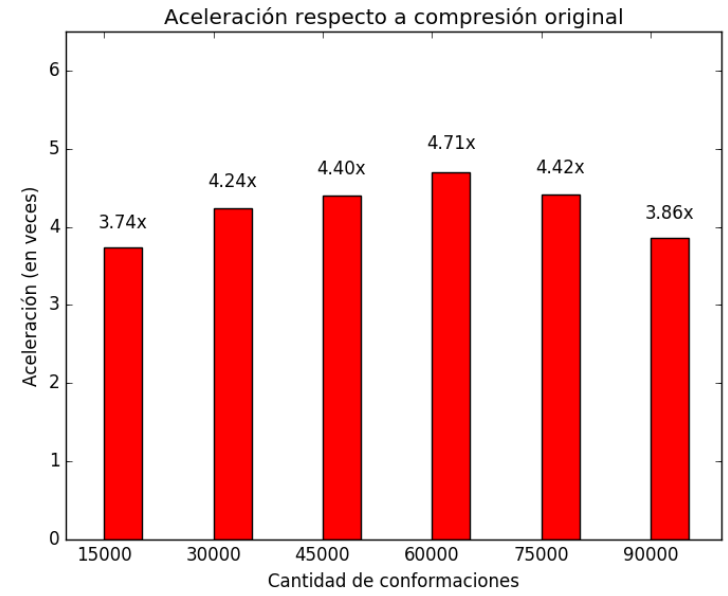

Fig. 4.8: Aceleración a medida que crece la cantidad de conformaciones respecto al programa original con un núcleo

En la figura 4.9 podemos observar que la mejora continúa siendo notoria, aún en el contexto de estar repartiendo carga en tres núcleos. Este resultado también contrasta con lo obtenido en la figura 4.6.

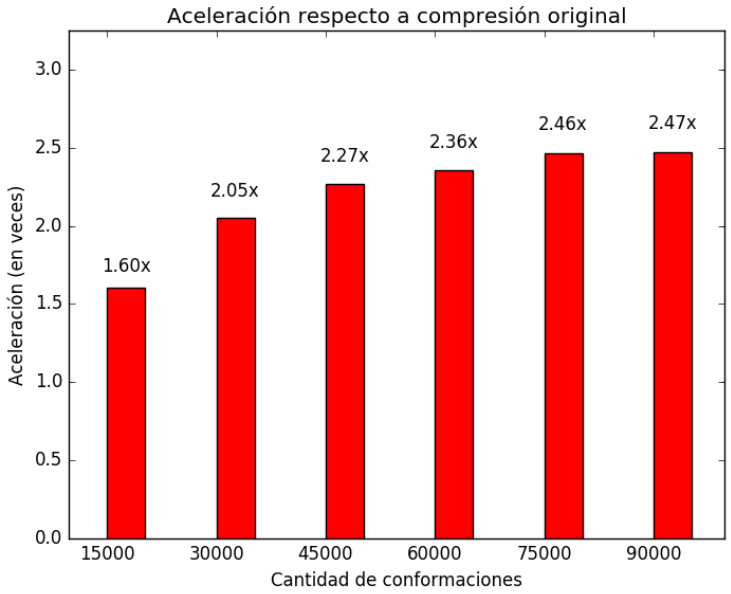

Fig. 4.9: Aceleración a medida que crece la cantidad de conformaciones respecto al programa original con tres núcleos

4.3. Resultados experimentales en computadoras de alto rendimiento

Decidimos experimentar con computadoras de prestaciones más potentes que la computadora usada para las pruebas anteriores, con el objetivo de verificar cuán escalable es el programa en estos entornos.

4.3.1. Computadora Intel I7 de 8 núcleos

En la Tabla 4.2 mostramos la configuración de hardware y software en la que se ejecutó el experimento

Sistema operativo	*Ubuntu SMP 16.04*, kernel *4.4.0-165-generic*
Solver	Sundials Kinsol *2.8.1*
Compilador	GNU Fortran compiler *5.4.0*
Flags del compilador	-O3
CPU	Intel(R) Core(TM) i7-7700 CPU @ 3.60 GHz, 8 cores
RAM	64 GB
HDD	WD Blue 1 TB Desktop Hard Disk Drive - 7200 rpm SATA 6 Gbit/s 64 MB Cache

Tab. 4.2: Computadora Intel I7 de cuatro núcleos

En la Figura 4.10 se puede observar la gran mejora respecto a la implementación original, sobre todo a partir de cuatro núcleos. En tamaños de problema elevados no se ve una gran diferencia de rendimiento entre cuatro y ocho núcleos, lo que creemos que ocurre debido a que hay cuatro núcleos físicos y ocho lógicos (derivados del *Hyperthreading* de Intel)

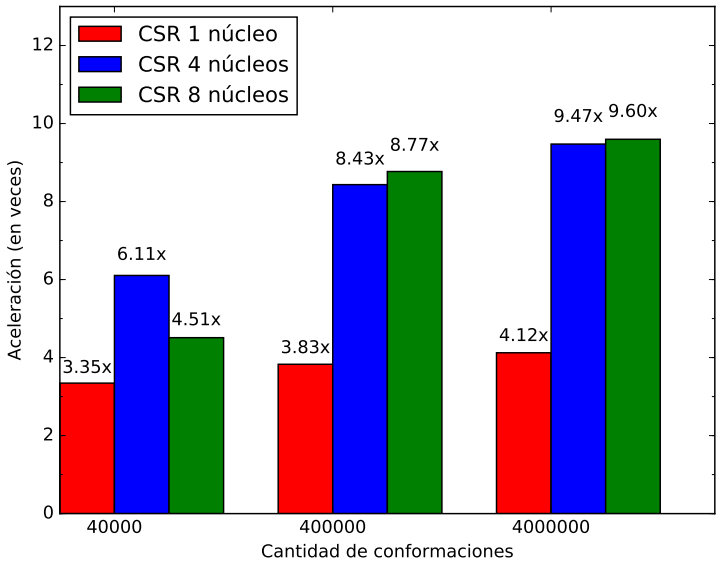

Fig. 4.10: Aceleración respecto a implementación original a medida que se incrementa la cantidad de conformaciones. La comparación es con respecto al programa original, utilizando en la versión CSR uno, cuatro y ocho núcleos.

4.3.2. Nodo individual de cluster TUPAC

En la Tabla 4.3 mostramos la configuración de hardware y software en la que se ejecutó el experimento

Sistema operativo	SMP Debian 4.19.28-2 x86_64
Solver	Sundials Kinsol *2.8.1*
Compilador	GNU Fortran (Debian 8.2.0-13) *8.2.0*
Flags del compilador	-O3
CPU	4 x AMD Opteron 6276 (16 cores) @ 2.3 GHz
RAM	126 GB
HDD	Seagate 7200RPM SATA 6Gb/s 500 GB
Networking	Mellanox MT26428

Tab. 4.3: Nodo de cluster TUPAC

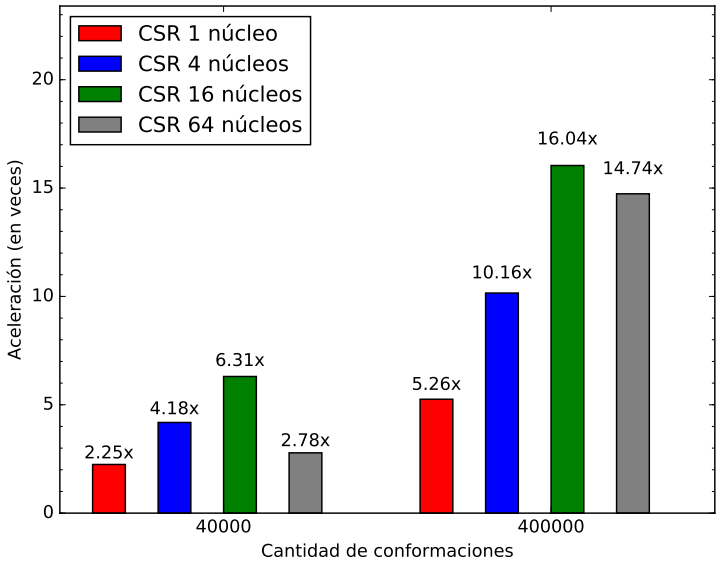

Fig. 4.11: Aceleración respecto a la implementación original a medida que se se incrementa la cantidad de conformaciones. La comparación es con respecto al programa original, utilizando en la versión CSR uno, cuatro, dieciséis y sesenta y cuatro núcleos. Puede observarse un decaimiento notorio de la performance cuando se utilizan más de treinta y dos núcleos, lo que atribuimos al *overhead* producido por tener que acceder a la memoria entre los procesadores.

4.4. ¿Por qué se logró mejor performance?

Viendo los resultados obtenidos, una pregunta que cabe hacer es: ¿Por qué el programa modificado resultó más rápido que la implementación original? A nuestro entender, esto se logró porque los esfuerzos de optimización estuvieron concentrados en lograr una mayor velocidad de ejecución de la porción del programa que identificamos como *código crítico*, reemplazando la estructura original por otra con resultados ya probados.

4.5. ¿Qué desventajas hay en utilizar la MKL de Intel?

- La biblioteca no implementa eficientemente el uso de múltiples procesadores de manera nativa, por lo que implica horas de trabajo para lograr exprimir al máximo la capacidad de cómputo de un sistema multicore.
- La biblioteca, a pesar de implementar las interfaces `BLAS` y `sparse BLAS` [29], es de código cerrado, lo cual complicó de manera sustancial el *debugging* durante la implementación.
- El problema más importante es que los resultados de las multiplicaciones matriz-vector difieren numéricamente de los obtenidos al utilizar la función `MATMUL` nativa de Fortran. Dicha diferencia, del orden de $2*10^{-16}$, provocaba que hubiera diferencias numéricas sutiles en las aproximaciones del programa **nanopore**; como consecuencia de dicha diferencia, el algoritmo de aproximación utilizado por la biblioteca Kinsol toma decisiones distintas, que llevan a un resultado $\mathbf{x}$ para $F(\mathbf{x}) = 0$ distinto al original. Este problema no tiene importancia en nuestra aplicación, pero podría tenerla en otras aplicaciones.

5. CONCLUSIONES Y TRABAJO A FUTURO

Se identificaron las falencias del programa original, la sección del código que contribuía en gran parte al tiempo de ejecución del mismo y la naturaleza matricial de las estructuras de datos utilizadas. Se identificó, en consecuencia, que dicha matriz era de naturaleza rala y que se precisaba una solución que aprovechara dicha naturaleza para lograr una mejor performance. Se utilizó una biblioteca para atacar el problema, y mediante adaptación del código se logró incrementar la performance obtenida mediante uso de los procesadores, logrando así el objetivo principal de la tesis. Adicionalmente, se probó su escalabilidad en un cluster, lo que abre la puerta a resolver problemas en el marco de la teoría molecular de mayor complejidad.

Además, aprendimos que al utilizar bibliotecas para resolver problemas numéricos, es muy importante que la documentación (tanto oficial como la información obtenible a través de foros, etc.) sea lo más completa posible, de modo tal que se pueda implementar la solución de la mejor manera posible. A su vez, es importante que dicha documentación sea clara, para lograr una integración multidisciplinaria satisfactoria entre las ciencias de la computación y, en este caso, las ciencias químicas.

Como trabajo futuro, creemos que se puede explorar la utilización de GPU, aunque creemos que para ello el código del programa debería ser reformulado desde sus cimientos.

Bibliografía

[1] I. Szleifer and M. A. Carignano, "Tethered Polymer Layers," 2007.

[2] R. Nap, P. Gong, and I. Szleifer, "Weak polyelectrolytes tethered to surfaces: Effect of geometry, acid-base equilibrium and electrical permittivity," *Journal of Polymer Science, Part B: Polymer Physics*, 2006.

[3] D. Frenkel and B. Smit, *Understanding molecular simulation: From algorithms to applications*. 1996.

[4] P. G. de Gennes and T. A. Witten, " Scaling Concepts in Polymer Physics ," *Physics Today*, 1980.

[5] G. Zaldivar, S. Vemulapalli, V. Udumula, M. Conda-Sheridan, and M. Tagliazucchi, "Self-Assembled Nanostructures of Peptide Amphiphiles: Charge Regulation by Size Regulation," *The Journal of Physical Chemistry C*, 2019.

[6] G. Zaldivar, M. B. Samad, M. Conda-Sheridan, and M. Tagliazucchi, "Self-assembly of model short triblock amphiphiles in dilute solution," *Soft Matter*, 2018.

[7] F. M. Boubeta, G. J. Soler-Illia, and M. Tagliazucchi, "Electrostatically Driven Protein Adsorption: Charge Patches versus Charge Regulation," *Langmuir*, 2018.

[8] F. M. Gilles, F. M. Boubeta, O. Azzaroni, I. Szleifer, and M. Tagliazucchi, "Modulation of Polyelectrolyte Adsorption on Nanoparticles and Nanochannels by Surface Curvature," *Journal of Physical Chemistry C*, 2018.

[9] G. Zaldivar and M. Tagliazucchi, "Layer-by-Layer Self-Assembly of Polymers with Pairing Interactions," *ACS Macro Letters*, 2016.

[10] I. Gleria, E. Mocskos, and M. Tagliazucchi, "Minimum free-energy paths for the self-organization of polymer brushes," *Soft Matter*, 2017.

[11] O. Peleg, M. Tagliazucchi, M. Kröger, Y. Rabin, and I. Szleifer, "Morphology control of hairy nanopores," *ACS Nano*, 2011.

[12] P. J. Flory, "Statistical Mechanics of Chain Molecules," *Physics Today*, 1970.

[13] F. Faggin, M. E. Hoff, S. Mazor, and M. Shima, "History of the 4004," *IEEE Micro*, 1996.

[14] E. Track, N. Forbes, and G. Strawn, "The End of Moore's Law," 2017.

[15] Y. Li and L. B. Kish, "Heat, speed and error limits of Moore's law at the nano scales," *Fluctuation and Noise Letters*, 2006.

[16] M. J. Flynn, "Some computer organizations and their effectiveness," *IEEE Transactions on Computers*, 1972.

[17] S. Oberman, G. Favor, and F. Weber, "AMD 3DNow! technology: Architecture and implementations," *IEEE Micro*, 1999.

[18] J. L. Hennessy and D. a. Patterson, *Computer Architecture, Fourth Edition: A Quantitative Approach*. 2006.

[19] T. Ungerer, B. Robič, and J. Šilc, "A survey of processors with explicit multithreading," 2003.

[20] R. Farber, *CUDA Application Design and Development*. 2011.

[21] J. E. Stone, D. Gohara, and G. Shi, "OpenCL: A parallel programming standard for heterogeneous computing systems," *Computing in Science and Engineering*, 2010.

[22] Y. Lecun, Y. Bengio, and G. Hinton, "Deep learning," 2015.

[23] R. L. Graham, T. S. Woodall, and J. M. Squyres, "Open MPI: A Flexible High Performance MPI," *Proceedings 6th Annual International Conference on Parallel Processing and Applied Mathematics*, pp. 228–239, 2005.

[24] A. C. Hindmarsh, P. N. Brown, K. E. Grant, S. L. Lee, R. Serban, D. E. Shumaker, and C. S. Woodward, "SUNDIALS: Suite of Nonlinear and Differential/Algebraic Equation Solvers," *ACM Transactions on Mathematical Software*, vol. 31, no. 3, pp. 363–396, 2005.

[25] P. J. Denning, "The locality principle," *Communications of the ACM*, vol. 48, no. 7, p. 19, 2005.

[26] GNU-Project, "Options That Control Optimization." \url{https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html}.

[27] N. CAPS Enterprise, CRAY Inc, The Portland Group Inc (PGI), "The OpenACC Application Programming Interface," *Openacc.Org Documentation*, 2011.

[28] S. C. Eisenstat, M. C. Gursky, M. H. Schultz, and A. H. Sherman, "Yale sparse matrix package I: The symmetric codes," *International Journal for Numerical Methods in Engineering*, 1982.

[29] Intel, "Intel ® Math Kernel Library Reference Manual," *Numerical Algorithms*, 2007.