EIA : Plantilla sencilla y cutre tipo artículo, una columna. Acentos disponibles.

Author: All of us

Wednesday, 29th March

1 Introduction

2 Initialisation of the system

3 Forces Elena

4 Forces Xabi

5 Integrador de Euler

Una de las principales partes de una código de dinámica molecular son los integradores de las ecuaciones de movimiento, aquellos que permiten la propia dinámica.

Para hacer un buen código de dinámica molecular, es menester elegir correctamente qué integrador se va a usar en la simulación. En esta sección se va a explicar el algoritmo de Euler explícito, uno de los más sencillos que se pueden implementar.

El algoritmo de Euler discretiza e integra las ecuaciones de movimiento Newton, (1), con tal de obtener la trayectoria de las partículas del sistema simulado.

$$\frac{\mathrm{d}\mathbf{r_k}}{\mathrm{d}t} = \mathbf{v_k} \qquad\qquad \frac{\mathrm{d}\mathbf{p_k}}{\mathrm{d}t} = \mathbf{F_k}. \tag{1}$$

Dado que se trata de una dinámica molecular, el integrador implementará la actualización de las posiciones y la de las velocidades a cada paso de tiempo, siguiendo las ecuaciones (2) y (3),

$$\mathbf{r_k}(t+\Delta t) = \mathbf{r_k}(t) + \mathbf{v_k}(t) + \frac{1}{2m_k}\mathbf{F_k}(t)\Delta t^2, \tag{2}$$

$$\mathbf{v_k}(t+\Delta t) = \mathbf{v_k}(t) + \frac{1}{m_k}\mathbf{F_k}(t)\Delta t. \tag{3}$$

No obstante, este integrador presenta un problema severo. Dicho problema es la falta de simetría bajo una transformación temporal inversa para Δt suficientemente grandes. Esto implica que si se quiere implementar un buen algoritmo integrador, habría que tomar pasos de tiempo infinitesimalmente pequeños al usar el integrador de Euler.

La implementación numérica de este método en un código de DM es sencilla. En cada paso de tiempo, una vez las fuerzas que actúan sobre cada partícula hayan sido calculadas, se añade este algoritmo para calcular las nuevas posiciones y velocidades.

5.1 Código en paralelo

El método usado para paralelizar las dos subrutinas que he implementado, los integradores para las posiciones y para las velocidades, ha sido la herramienta MPI para Fortran90.

Con tal de paralelizar el cálculo del update de posiciones y velocidades, he elegido dividir los vectores según el índice correspondiente al número de partículas, i.e., que cada procesador ejecute el cálculo de un segmento de partículas, $[k, k + \delta k]$. El número de partículas que entre en cada procesador vendrá dado por el número de procesadores y el número de partículas, e.g, en el caso de 12 partículas y 4 procesadores, cada uno ejecutará los cálculos para 3 partículas.

Una vez cada procesador ha calculado su correspondiente segmento del vector, lo envían al Master (procesador 0), el cual hace un *merge* de todos los segmentos con tal de obtener el vector entero de posiciones y velocidades ya actualizado a ese instante de tiempo. Cuando todo el vector está ya actualizado, el Master vuelve a mandar el vector a todos los procesadores, consiguiendo así que todos ellos tengan toda la información actualizada.

Para enviar datos entre procesadores, he usado la subrutina del módulo 'mpif.h' MPI_ISEND. Una vez los datos han sido enviados, antes de ser recibidos, he implementado una barrier, MPI_BARRIER, con tal de esperar a que todos los procesadores hallan enviado su sección del vector y evitar fallos en la comunicación. Cuando la barrera ya ha sido superada, los destinatarios reciben los datos con un MPI_RECV, seguido también por una barrera que sólo sea superada una vez todos los destinatarios hayan recibido la información.

Para testear la paralelización, he estudiado la dependencia del tiempo de ejecución de mis subrutinas (posiciones y velocidades por Euler) en función del número de procesadores y del número de partículas tratadas. Como se puede ver en los resultados mostrados en la Figure 1, la paralelización no ha sido efectuada de modo eficiente, pues cuantos más procesadores, más lenta es la ejecución. El mismo comportamiento ha sido observado para el caso del integrador de las velocidades.

También se puede observar en la misma figura que el número de partículas tratadas tampoco influye significativamente en el tiempo de ejecución. Todo ello lleva a pensar que el método usado no ha sido el óptimo, y que la paralelización efectuada no ha servido para acelerar el código. Un posible motivo es la simplicidad de las subrutinas, que sólo consistían en un único bucle que

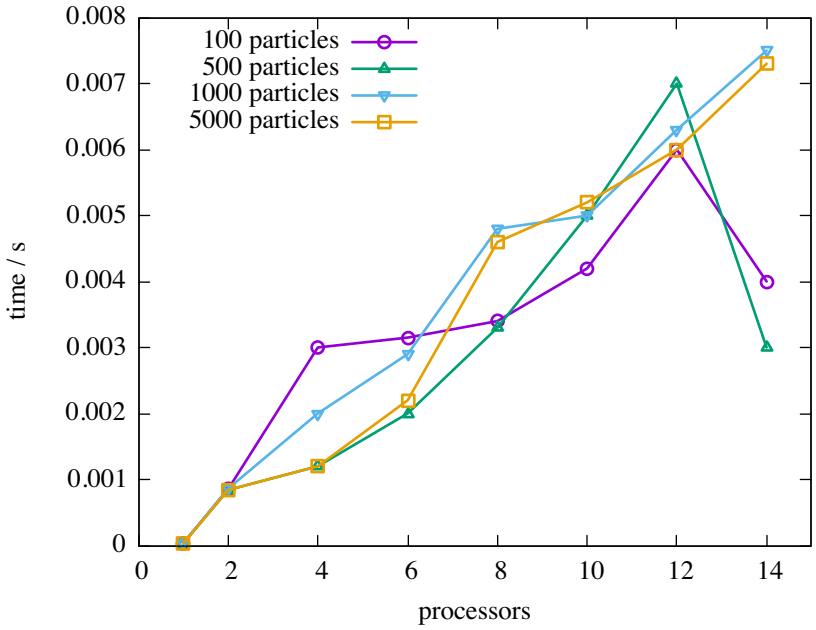

Figure 1: Dependencia del tiempo de ejecución del integrador Euler para las posiciones con el número de procesadores y partículas.

recorría todo el vector de posiciones. Bucles de esta índole no son lentos, y escalan correctamente, mientras que el hecho de implementar la comunicación entre los procesadores relentiza el código. Cuantos más procesadores, más comunicaciones habrá, y más aumenta el tiempo de ejecución, tal y como se muestra en la Figure 1.

6 Integrador Velocity Verlet

El segundo integrador usado en el código es velocity Verlet y se aplica usando las siguientes expresiones dependiendo de la variable a integrar:

$$\mathbf{r}(t+\Delta t) = \mathbf{r}(t) + \mathbf{v}(t)\Delta t + \frac{\mathbf{F}(t)}{2m}\Delta t^2, \tag{4}$$

$$\mathbf{v}(t+\Delta t) = \mathbf{v}(t) + (\mathbf{F}(t) + \mathbf{F}(t+\Delta t))\,\frac{\Delta t}{2m}. \tag{5}$$

Para calcular las posiciones se usa exactamente la expresión de Velocity Verlet pero para las velocidades está estructurado en dos partes. Primero se calcula la velocidad en el paso de tiempo actual, luego se calculan las fuerzas para actualizar la aceleración de las partículas y finalmente, con la nueva aceleración, se calcula el término a tiempo posterior. El programa tiene esta estructura para

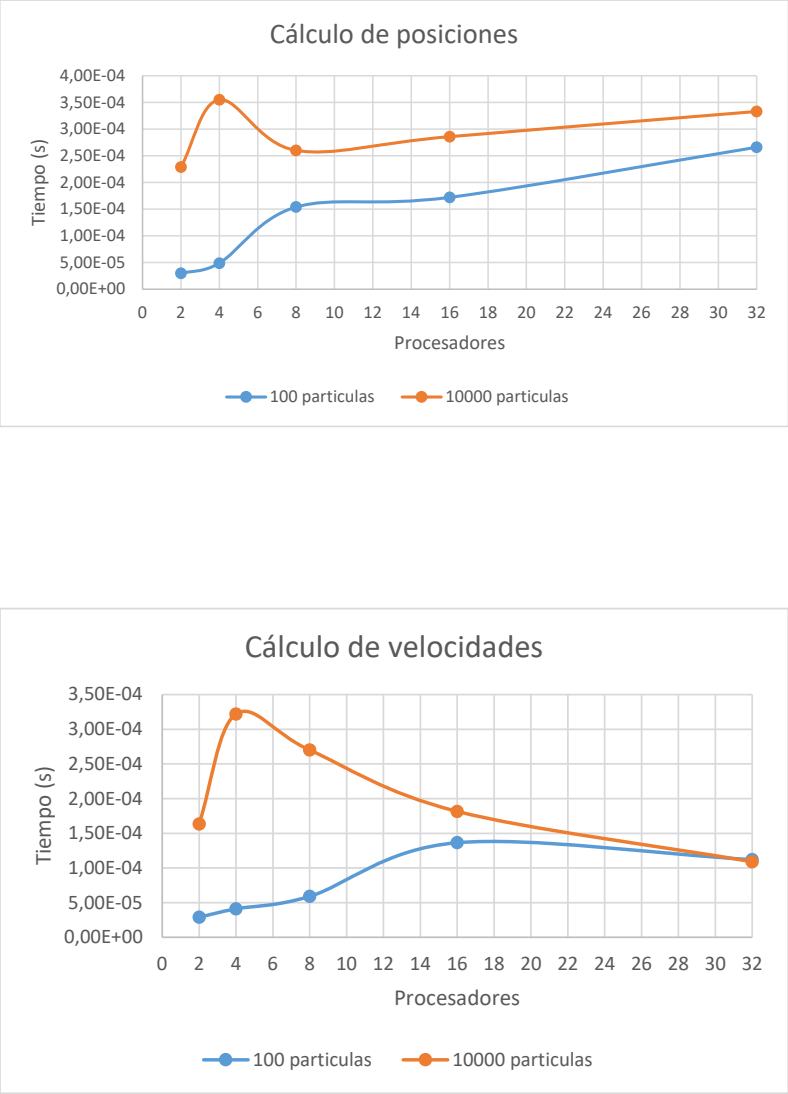

Figure 2: Dependencia del tiempo de ejecución del integrador Verlet para las posiciones y velocidades con el número de procesadores y partículas.

no tener que guardar la aceleración en el tiempo anterior para cada partícula, lo que reduce la memoria que necesita el programa.

Al realizar el cálculo en paralelo primero se realiza una parte del cálculo en cada uno de los procesadores a los que se ha enviado el cálculo. El reparto de los datos en cada procesador se realiza en el módulo paralelizar e importamos las variables a nuestro módulo. Luego enviamos la parte calculada de cada worker al master usando mpi_isend e introducimos una barrier para asegurar-nos de que todos los workers han enviado su parte antes de seguir con el programa. Entonces usamos un mpi_recv para que el master reciba todos los datos enviados.

Finalmente, el master envía el vector de datos entero actualizado a todos los workers usando también un mpi_isend, mpi_barrier y mpi_recv.

Una vez comprobado que el programa funciona correctamente realizamos un estudio para comprobar como escala el tiempo de ejecución de las subrutinas respecto al número de procesadores para dos cantidades de partículas diferentes (100 y 10.000).

Teóricamente esperaríamos el mismo comportamiento para las dos subrutinas, ya que solo difieren en el cálculo a realizar, pero no es exactamente lo que se observa. Para 100 partículas vemos en los dos casos que el tiempo de ejecución aumenta al aumentar el número de procesadores de una forma más o menos lineal, exceptuando uno de los puntos en cada caso. Para 10.000 partículas se observa un máximo de tiempo al usar 4 procesadores y luego en el caso de las velocidades el tiempo disminuye al aumentar los procesadores y para las posiciones primero disminuye y luego aumenta ligeramente.

Esta diferencia entre las dos subrutinas puede ser debida a que algunos cálculos se han realizado en nodos diferentes, por lo que no se puede garantizar que el tiempo de cálculo sea el mismo para cada ejecución del programa. Para estudiar rigurosamente la escalabilidad del programa se requeriría de más tiempo y más control sobre los procesadores donde se ejecuta el programa.

7 Condiciones periódicas de contorno

Las condiciones periódicas de contorno nos aseguran que si la partícula se encuentra fuera de la caja nuestra subrutina encauza a ésta de manera que se halle en la posición de convención de imagen mínima. Esto implica que, si la partícula sale por un lado de la celda, entrará por el lado opuesto al de salida (Figure 3). Estas son necesarias para simular sistemas grandes en un espacio más reducido de éste.

Para aplicar esto, el código sitúa el origen de coordenadas en el centro de nuestra celda de simulación. Entonces, debemos restar una longitud proporcional al tamaño de la caja (L) a aquellas partículas que se encuentren más allá de L/2 en cualquiera de las tres direcciones, en caso de 3D (Figure 4). De manera que, la constante de proporcionalidad venga dada por el número entero más cercano resultante de la división entre la posición de dicha partícula y L. En caso de que esta se encuentre en el interior de la celda esta constante valdrá 0 y la posición de la partícula no será variada. Para realizar este redondeo en

Fortran90 se emplea la función nint, quedando el código como:

$$\mathbf{r}(i, \text{dimensión}) = \mathbf{r}(i, \text{dimensión}) - L \cdot \text{nint}\left(\frac{\mathbf{r}(i, \text{dimensión})}{L}\right), \qquad (6)$$

dónde i recorrería de uno a N (número total de partículas), y dimensión las diferentes coordenadas para el eje x, y, z.

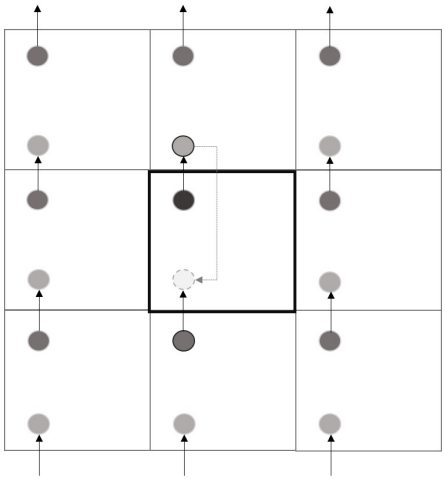

Figure 3: Representación de la convención de imagen mínima para una lattice 2D.

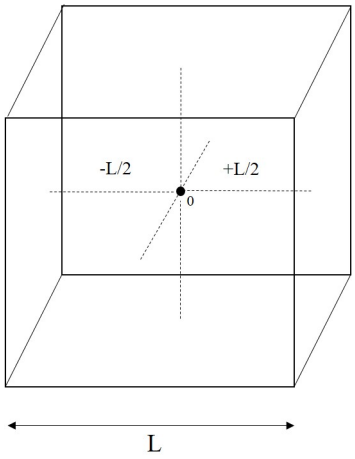

Figure 4: Celda de simulación 3D centrada en el origen de coordenadas.

7.1 Paralelización MPI

Para paralelizar la subrutina haremos la inicialización y finalización de MPI en el código principal incluyendo el módulo 'mpif.h', y repartiremos el número de partículas según el número de procesadores que empleemos, con los vectores ini y fin. Haciendo que, si no hay un número entero de partículas, el último procesador haga una partícula de más. Entonces en la subrutina, los diferentes workers con un MPI_ISEND enviarán sus posiciones ya modificadas de las partículas que les pertenecen al master (procesador 0). El master con un MPI_RECV recibirá toda la información actualizada y enviará toda esta con un MPI_ISEND a todos los workers. Estos también la recibirán con MPI_RECV, consiguiendo así que todos dispongan de las nuevas posiciones ya modificadas.

Entre estos procesos de comunicación entre procesadores encontraremos llamadas a las subrutinas del módulo mpi que crean las barreras, MPI_BARRIER. Estas obligan a esperar a tener toda la información de todos los workers o master antes del siguiente proceso de comunicación.

Para ver el efecto de la paralelización de dicha subrutina se ha graficado el tiempo que tarda esta con un número concreto de partículas (N=10000 y N=100) variando el número de procesadores empleados, obteniendo,

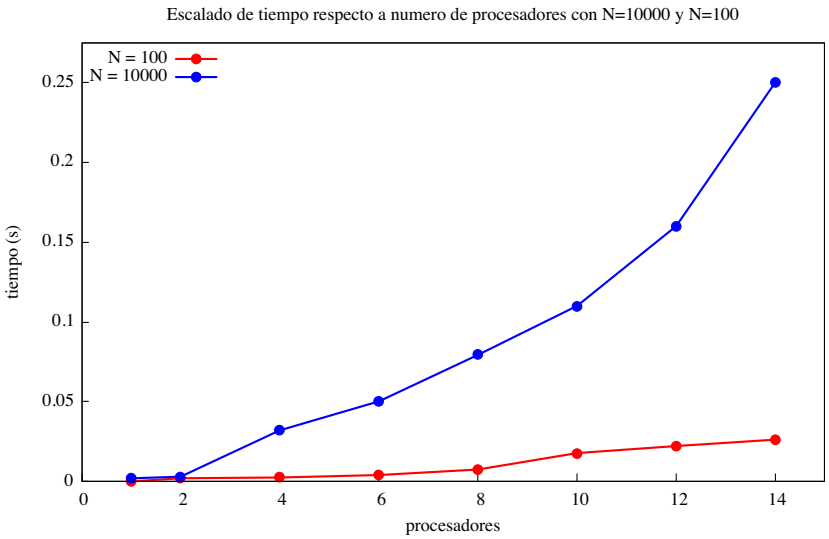

Figure 5: Dependencia del tiempo de ejecución de las condiciones periódicas de contorno con el número de procesadores a un número de partículas constante.

Se puede observar que en este caso particular tarda más cuanto más procesadores haya. Esto es porque lo que ralentiza el proceso no es la línea de código de condiciones periódicas, si no el proceso de comunicación entre procesadores. Observaríamos, la ley de Amdahl's pero a la inversa con N=10000. Este efecto se observa con mayor número de partículas. En este caso, no aumenta el tiempo

linealmente con el número de procesadores porque siempre tenemos una parte serial de código.

También se ha estudiado como varia el tiempo de cálculo en función del número de partículas manteniendo un número de procesadores constante (procesadores=4).

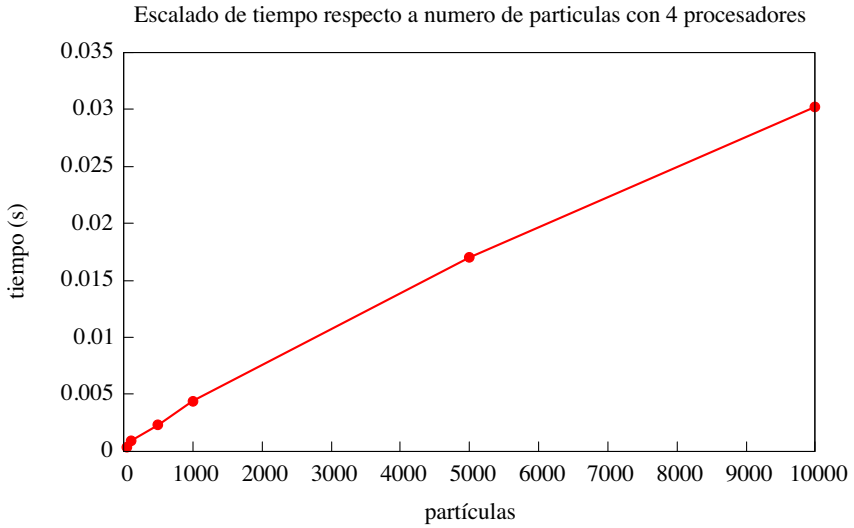

Figure 6: Dependencia del tiempo de ejecución de las condiciones periódicas de contorno con el número de partículas a un número de procesadores constante.

Como es lógico a mayor número de partículas se observa un mayor tiempo de cálculo. Y se observa una tendencia aceptablemente lineal.

La determinación de puntos en ambas gráficas son resultado del promedio de diferentes medidas del tiempo ya que el clúster utilizado presenta distintos nodos, no pudiéndose controlar a cuál de ellos envía. De este modo, el resultado no es reproducible de una ejecución a otra.

8 Data

9 Desplazamiento cuadrático medio

A continuación se detallará el procedimiento seguido en la implementación de la paralelización del código de análisis visual, última parte del código de dinámica molecular desarrollado.

Para esta parte se decide calcular el desplazamiento cuadrático medio (MSD) de las partículas a lo largo de la simulación, con la expresión:

$$MSD \equiv \langle (x(t) - x_o)^2 \rangle \tag{7}$$

Para calcular dicha expresión, se introduce en el archivo de entrada un valor de la frecuencia de escritura de las posiciones de las partículas (*restart*) y en el *main* del archivo que las contiene (con el mismo nombre y extensión *rst*). Del mismo modo, al iniciar la simulación se determina el número de frames que contendrá el archivo de restart.

Una vez finalizada la simulación, previamente a finalizar el entorno MPI, se llama a la subrutina *postvisual* con la entrada de número de frames, número de partículas y dimensión del sistema.

La subrutina (en serie y en paralelo) lee el archivo generado a lo largo de la dinámica molecular en una matriz. Posteriormente, mide para cada partícula el MSD a cada frame impreso. Es decir, para una misma partícula conociendo el número de frames y partículas del sistema recorre la matriz calculando la expresión (7). Dichos valores se van imprimiendo en una nueva matriz con F filas (correspondientes una a cada tiempo) y C columnas (una por cada partícula).

Finalmente, el resultado se imprime en un archivo con nombre *desplazamientos.dat*.

9.1 Implementación en paralelo

Se decide que la paralelización de la rutina se realice dividiendo las N partículas del sistema en los diferentes procesadores a usar. De este modo, se consigue que cada procesador pueda calcular el valor de MSD con la rutina ya generada para las partículas asignadas de manera autónoma, y sólo deba comunicarse con los demás al finalizar el cálculo.

Por otra parte, la lectura del archivo *restart.rst* se realiza también en paralelo. Éste hecho se debe a que en un primer momento se realizaba de manera única a través del MASTER y éste enviaba a cada procesador el rango correspondiente –según la división de partículas realizada– de la matriz de entrada para que realizara el cálculo. Se observaron problemas de comunicación por parte del MASTER a los procesadores y se decidió implementar también la lectura en paralelo.

Con la lectura en paralelo se han observado los siguientes resultados:

- Desaparición de los problemas de comunicación existentes, al no tener que comunicarse el MASTER con cada procesador.
- Tiempo de cálculo similar en ambos métodos, ya que la lectura del archivo se debía realizar de todos modos.

Se debe considerar que pese a que el tiempo de cálculo no aumenta, el gasto de memoria sí que incrementa debido a que la lectura del archivo *restart.rst* es realizada por todos los procesadores. En cualquier caso, se hace un balance positivo ya que en el método anterior intentado se habían producido problemas en algunos tests realizados a nivel de comunicación que no se habían podido solucionar.

Posteriormente, cada procesador calcula el MSD para las partículas asignadas en la división de trabajo realizada y mandan los rangos de la matriz de

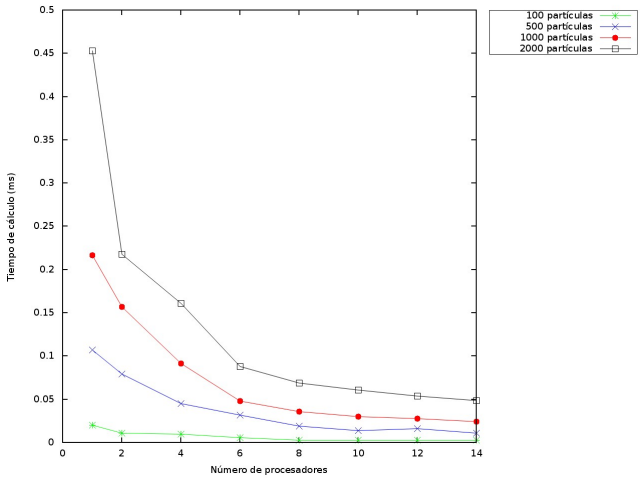

Figure 7: Tiempo de cálculo (ms) para diferentes sistemas (N=100, 500, 1000, 2000) en función de los procesadores utilizados.

desplazamientos a MASTER. Éste, después de recibir los datos, los imprime en el archivo de resultados final.

Así, el proceso se puede resumir como:

1. Lectura del archivo de entrada completo en paralelo.
2. Cálculo del MSD en paralelo, con división por partículas en los diferentes procesadores.
3. Envío de valores de MSD de los diferentes procesadores a MASTER.
4. Impresión de los resultados en *desplazamientos.dat*.

9.2 Análisis de la implementación en paralelo

La implementación en paralelo se ha realizado de manera satisfactoria obteniendo unos resultados de disiminución de tiempo considerables en aumentar el número de procesadores.

Para diferente número de partículas (figura 7 se ha analizado el tiempo de cálculo de la rutina para un número creciente de procesadores (de 1 –en serie– a 14) observando como la disminución del tiempo de cálculo es considerable entre 2 y 6 procesadores, pero a partir de éstos el tiempo es muy parecido.

Se puede observar como para sistemas de mayor tamaño (N=1000,2000) la disminución de velocidad entre el cálculo en serie y con dos procesadores es remarcable. Prácticamente, con 2000 partículas en aumentar el número de procesadores en dos se consiguen tiempos de cálculo comparables al sistema de 1000 partículas con dos procesadores menos. A partir de 8 procesadores dicha tendencia se pierde obteniendo un comportamiento asintótico.

Analizando el tiempo de cálculo para el mismo número de procesadores en aumentar el de partículas obtenemos un incremento lineal del tiempo, tal y como se puede observar en la figura 8. En este caso, se puede observar una

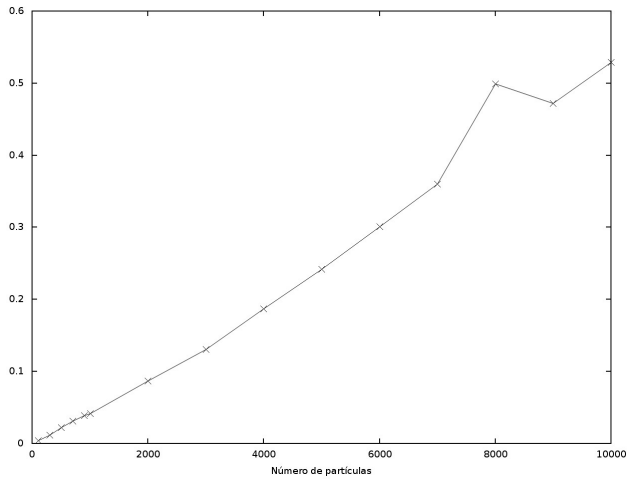

Figure 8: Tiempo de cálculo (ms) para seis procesadores en función del tamaño del sistema.

dependencia cuasi-lineal respecto el tiempo de cálculo y el tamaño del sistema.

9.3 Conclusiones

Del proceso de paralelización de la rutina de cálculo del desplazamiento cuadrático medio de las partículas, podemos concluir:

- El proceso de paralelización a través de la división por partículas permite disminuir el tiempo de cálculo de manera relevante en aumentar los procesadores utilizados, equiparando la velocidad a sistemas más pequeños para un número de procesadores menor o igual a 6.
- Dicha disminución del tiempo de cálculo no se observa para un número de procesadores mayor a 6, ya que se observa un comportamiento asintótico.
- Para un mismo número de procesadores, hay una relación cuasi-lineal entre el tiempo de cálculo y el tamaño del sistema.
- La paralelización, pues, es efectiva para pocos procesadores y permite calcular sistemas 4 veces mayores en el mismo tiempo (veáse figura 7, la velocidad para el sistema con 2000 partículas con 6 procesadores es similar al cálculo en serie con 500 partículas).