Práctica 1

Programa Secuencial

Introducción

En esta práctica se debe de realizar un programa que calcule la energía (E) y la energía acumulada (T) en una superficie NxN por con P partículas, cada una con una energía y una posición en la que impacta diferente. El objetivo de este laboratorio es hacer un programa secuencial, y estudiar su tiempo teórico en FLOPS y el tiempo real en segundos (con y sin la opción -O2). El programa debe de escribirse en C o C++, y en posteriores sesiones será optimizado con Open MP y MPI.

Para realizar este programa se ha optado por usar C++, pero no he usado ninguna estructura he hecho uso de librerías como std::vector para alojar memoria dinámica, con el fin de hacer más fácil su implementación paralela en un futuro. Respecto a las características usadas de C++ que no posee C, únicamente he usado la librerías para mostrar por pantalla y el operador new.

A continuación se relata como se ha programado en detalle la simulación. Pero el programa básicamente

Alojamiento en memoria

Para el alojamiento en memoria, antes de ejecutar se llama a la función **init_arrays**, que inicializan todas las variables que utilizan memoria dinámica.

Para las variables de entrada se ha optado por 3 punteros, que apuntarán al inicio de un array dinámico que cada que cada uno de ellos representa una propiedad de la partícula representada por un índice (x para la posición en un eje, y para la posición en otro eje y e para la energía de dicha partícula). Se ha elegido este modo porque es el más escueto a la hora de escribir los cálculos.

Para la salida T es un puntero que apunta a un array que cada elemento representa una partícula. Respecto a E, es una matriz de memoria continua (esto se ha realizado así para facilitar la futura paralelización en MPI). Su implementación consiste en la creación de un array NxN y posteriormente la creación de un array de punteros que apuntan a la primera posición de cada columna de la matriz (cada puntero apunta a N posiciones posteriores a la anterior, comenzando por la primera posición de memoria), la posiciones de la matriz se crean inicialmente con un malloc que las contenga a todas, y posteriormente se asignan los punteros.

Las varibales de entrada y salida son globales para evitar tener que pasarlas en cada parámetro y hacer el código más legible y eficiente computacionalmente.

Cálculo de T y E

$$E(x,y) = \sum_{p=1}^{P} EA(p,x,y), \qquad T(p) = \sum_{x=0}^{N-1} \sum_{y=0}^{N-1} EA(p,x,y)\,.$$

La **función calc** es la que se encarga de todos los cálculos del programa (calcular todos los valores de T y E). Hay tantos T como partículas y cada uno de los resultados de T se calcula con el

sumatorio EA para todas las posiciones del espacio posible. En cambio, respecto a E, existen tantos resultados como puntos de la superficie y se calcula con el sumatorio de cada partícula (por eso es coste es cuadrático respecto a N y lineal respecto a P) . Por tanto, se puede llegar a la conclusión que solo hace falta calcular una vez EA para cada partícula y posición del espacio y sumarlos a su respectivos resultados de E y T. Para evitar repetir el cálculo del valor EA se crea la variable EA_pxy_, que almacena temporalmente un resultado EA para una particula con en un punto determinado. Para ello se ha realizado un bucle que recorre todos los valores de P y de la superficie NxN, donde dentro se cálculo el valor EA que se añade a la partícula y la posición del espacio de dicha iteración. Para el cálculo de EA se ha creado una función, que se explicará su implantación más tarde.

Los valores de de E y T son inicializados previamente a 0. Para T se aprovecha las iteraciones del bucle exterior, pero para E se ha debido crear un doble bucle para incializarlo, ya que en cada iteración del bucle principal del cálculo no queremos reiniciar el valor de E. Posteriormente en cada.

También se calcula la variable global N2 que contiene el valor de N^2 para que no tenga que ser calculado múltiples veces. Posteriormente, dicha variable será utilizada en EA. Se ha utilizado ha optado por multiplicar N*N y mantenerla como entero con el fin de ahorrar FLOPS.

Cálculo de EA

$$EA(p,x,y) = \frac{\epsilon_p}{A(p,x,y) \cdot N^2},$$

EA es la energía acumlada de la particula p respecto al punto (x,y). Para no calcular N*N, se crea la variable N2, que se inicializa en la función anterior. Se ha utilizado una función A para calcular el factor de atenuación. Su implementación realiza los mismos cálculos que la expresión en el mismo orden.

Cálculo de A

$$A(p,x,y) = \exp\left(\sqrt{(x_p - x)^2 + (y_p - y)^2}\right)$$

A es el factor de atenuación para una partícula sobre la posiciones X e Y. Primero se ha de obtener la diferencia de distancia en cada eje respecto a la posición de la partícula y la que posición en la que se desea medir. ha sido necesario hacer un cast de cada variable a int, porque estaban definidas las variables como usnsigned y eso evita que se de el caso de underflow cuando el resultado sea negativo. Posteriormente dichas distancias en cada eje se elevan al cuadrado multiplicándose por si mismas y se suman y se saca la raíz cuadrada de dicha expresioón, con el con el fin de obtener la distanciae euclídea. Posteriormente se calcula la exponencial sobre la distancia euclídea expresión.

Coste del algoritmo

En este apartado se ha pasado todo el texto manuscrito a texto escrito a máquina. También se ha quitado la imagen del texto con el fondo oscuro porque contrastaba con el fondo.

El algoritmo **presenta un coste cuadrático respecto a N y lineal respecto a P**.

Tarda exactamente **25 · N² · P FLOP**. Algunas operaciones de suma y multiplicación no cuentan FLOPS porque se realizan con enteros.

Para llegar a dicho resultado hemos analizado los bucles y hemos hecho un sumario de todas las iteraciones , el interior va de Y=0 hasta Y=N-1, el exterior va de X=0 hasta N-1 y el exterior va de p=0 hasta p=P-1. Teniendo en cuenta que la función EA cuesta 25 (se explicará posteriormente como se ha llegado a dicha conclusión), se ha llegado a la conclusión tras usar la siguiente propiedad de los sumatorios de que el valor es **25 · N² · P FLOP.**

$$T(N,P) = \sum_{p=0}^{P-1} \sum_{x=0}^{N-1} \sum_{y=0}^{N-1}(25) = 25 \cdot \mathrm{N} \cdot \mathrm{N} \cdot \mathrm{P}$$

Para poder simplificar la expresión de arriba. Se ha utilizado esta expresión. Siendo es una expresión constante respecto a K.

$$\sum_{k=0}^{K-1}(q) = Kq$$

A continuación, se puede ver el cálculo de coste, donde se puede observar cuantos FLOP realiza cada sentencia. Las operaciones en las que solo intervienen números no reales no computan para el coste (como la inicialización de la variable dx y dy).

```
// Coste de función = 18FLOP
// 2 Multiplicaciones (1FLOP por operación)
// 1 Exponenicial (8FLOP por operación)
// 1 Raíz cuadrada (8FLOP por operación)
float A(const unsigned p, const unsigned X, const unsigned Y) {
    float dx = ((int) X) - ((int)x[p]);    dx *= dx;  // +1FLOP
    float dy = ((int) Y) - ((int)y[p]);    dy *= dy;  // +1FLOP
    return exp(sqrt(dx+dy)); // +8+8+1FLOP = 9 FLOP
}

// Coste de función = 25FLOP
// 1 Cáclulo de A (18FLOP por llamada)
// 1 Multiplicación (1FLOP por operación)
// 1 División (1FLOP por operación)
float EA(const unsigned p, const unsigned X, const unsigned Y) {
    return e[p] / ( A(p,X,Y)*N2 ); // 4+18+1FLOP
}

// Coste total = 25 * P * N * N = 25 P N^2 FLOP
void calc() {
    N2 = N*N;

    float EA_pxy_;

    for (unsigned X=0; X<N; X++)
        for (unsigned Y=0; Y<N; Y++)
            E[X][Y] = 0;
```

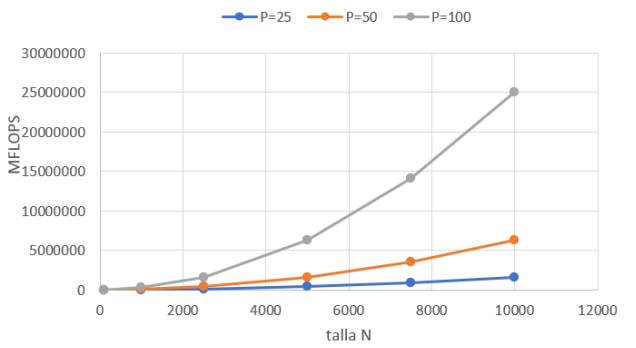
```
    for (unsigned p=0; p<P; p++) { // se repite P veces
        T[p] = 0;
        for (unsigned X=0; X<N; X++) // se repite N veces
            for (unsigned Y=0; Y<N; Y++) { // se repite N veces
                EA_pxy_ = EA(p,X,Y); // 25FLOP
                T[p] += EA_pxy_;
                E[X][Y] += EA_pxy_;
            }
    }
}
```

El calculo del coste se ha realizado teniendo en cuenta el anterior principio de matemático. Si sumas k veces (de 0 hasta k-1) un termino q, tal que q no es una expresión dependiente de k', el resultado es q veces k.

MFLOPS teóricos

Para el cálculo de los MFLOPS solo se debe de sustituir en el coste calculado los valores de N y P para los que se han realizado las pruebas y posteriormente dividir entre 10^6 (Ya que el coste original esta en FLOPS y cada MFLOP es un millón de FLOPS).

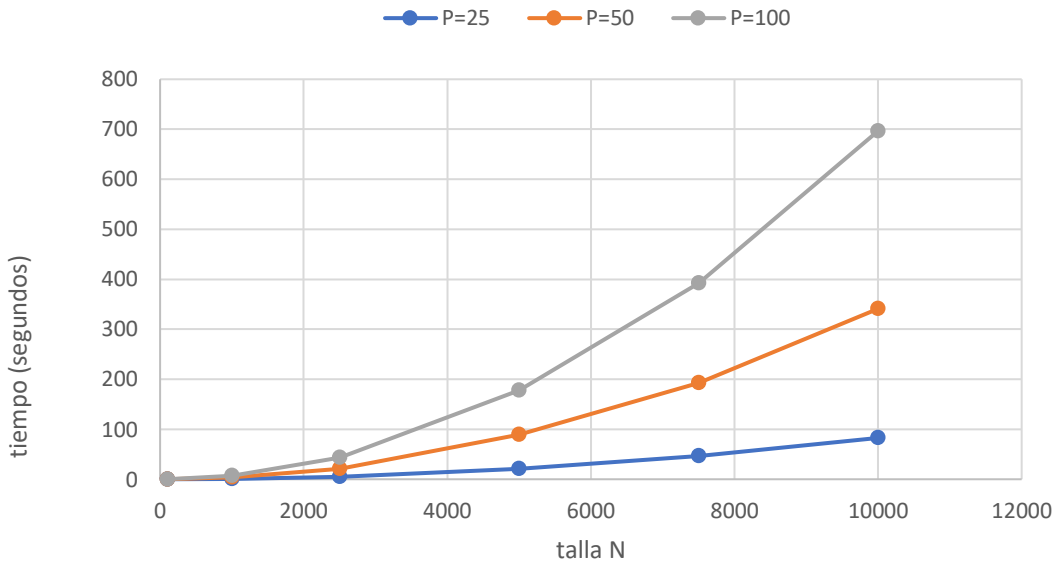
SECUENCIAL MFLOP			
N\P	25	50	100
100	6,25	12,5	25
1000	625	1250	2500
2500	3906,25	7812,5	15625
5000	15625	31250	62500
7500	35156,25	70312,5	140625
10000	62500	125000	250000



Tiempos experimentales sin optimización

Ejecutando el programa en la máquina boe.uv.es, para las diferentes tallas se obtienen los resultados de las tablas en segundos. Se puede observar que el orden de los resultados empíricos coincide con los teóricos, N aumenta de forma cuadrática y P lineal, pero lo veremos en profundad en el siguiente apartado, ya que los resultados con -O2 son más precisos.

N\P	25	50	100
100	0,0116181	0,0360079	0,0716174
1000	0,82673	3,4185	6,96296
2500	5,16641	21,3135	43,5681
5000	20,6559	89,1168	177,932
7500	46,444	192,221	392,335
10000	82,6465	340,96	696,621



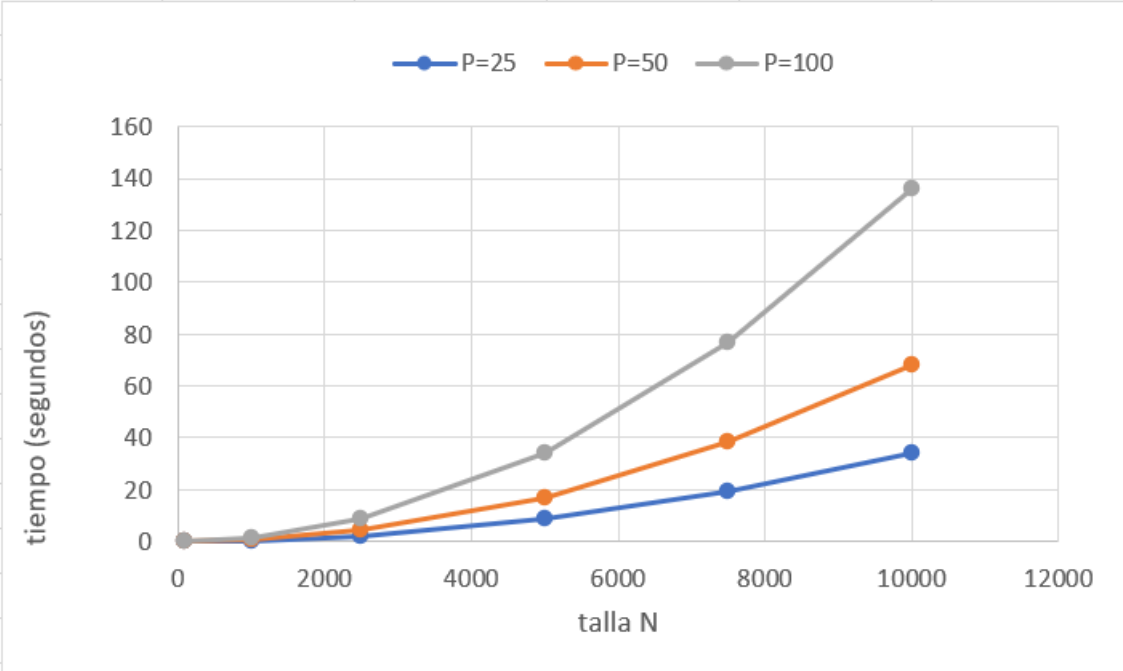
Tiempos experimentales con optimización -O2

La opción -O2 en el compilador de C/C++ analiza el código realizando optimizaciones del código para obtener una mejor velocidad de ejecución al ejecutar el programa sin afectar al espacio en memoria. Al aplicar esta opción al compilar el programa, al realizar los mismos experimentos se debería de notar una mejoría de tiempo de obtener mejores resultados, y en efecto así ha sido.

Tras realizar el experimento se ha observado que al aplicar la opción -O2, se obtienen resultados que coinciden en mayor medida con los resultados teóricos. Por tanto la observación de los resultados de va a realizar en este apartado.

Conforme se aumentan las tallas se ve con mayor precisión cuanto afecta N y P al coste temporal del algoritmo. Como se puede observar, si se dobla P el algoritmo tarda el doble evidenciando dependencia lineal respecto a dicha variable. Igual que sucede con P de forma linear, N al tener una dependencia cuadrática duplicar dicha variable, su coste temporal se cuatriplica.

N\P	25	50	100
100	0.00955129s	0.0151496s	0.0260112s
1000	0.364875s	0.706565s	1.38405s
2500	2.16167s	4.28355s	8.5334s
5000	8.55797s	17.0577s	34.0586s
7500	19.2255s	38.3685s	76.5814s
10000	34.1711s	68.1692s	136.149s



Rendimiento

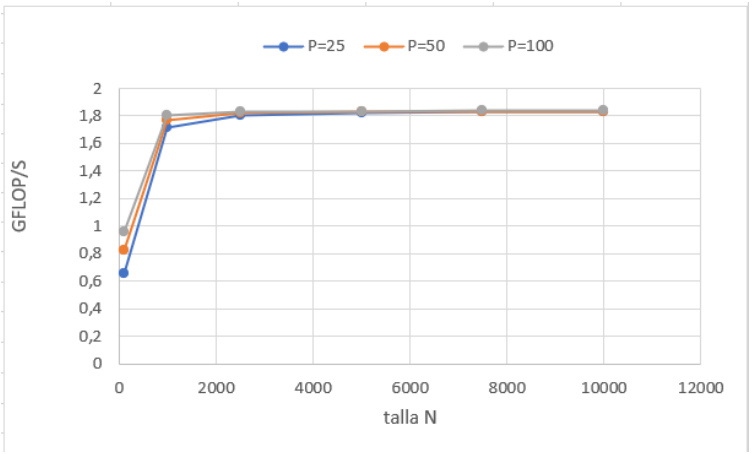
Para el cálculo del rendimiento se ha utilizado ha dividido el coste entre el tiempo, para obtenerlo en GFLOP/s se ha dividido entre mil, ya que el coste estaba en MFLOP y el tiempo en segundos. Este valor nos indica la cantidad de operaciones que realiza el programa por unidad de tiempo, por tanto es útil para determinar si nuestro programa esta utilizando la totalidad de los recursos de procesador o no. También cuando se desarrollen futuras versiones del programa que utilicen paralelismo nos va a ser útil esta medida como referencia, para ver como de eficaz ha sido la paralelización propuesta del algoritmo.

Observando los datos obtenidos, sin el -O2 se obtiene un rendimiento menor que aplicando dicha opción como es previsible esperar. No obstante los resultado de -O2 se ve que no utilizan toda la potencia del procesador con tallas N menores 1000, y ya a partir de 100 el rendimiento se mantiene constante pese seguir aumentando la N, dando entender el núcleo (en este caso solo uno porque en un programa secuencial) esta haciendo uso de toda su capacidad computacional. A partir de superar dicho umbral el número de operaciones por segundo son constantes y se aproxima a 1’8GFLOPS.

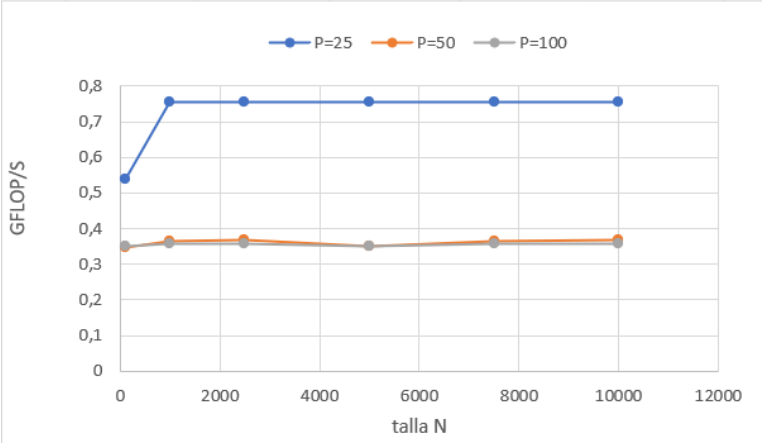
En el caso de no aplicar optimización las cantidad de GFLOP disminuye, obviamente, pero cuando la talla P=25 se puede observar que duplica el rendimiento de las tallas P=50 y P=100. Creo que

esto se debe a que como se han usado funciones dentro de la función calc(), la tasa de acierto del predictor de la caché aumenta si el tamaño de P es más pequeño.

RENDIMIENTO GFLOP/S (-O2)			
N\P	25	50	100
100	0,65436187	0,82510429	0,96112444
1000	1,71291538	1,76912244	1,80629313
2500	1,80705196	1,8238377	1,83104038
5000	1,82578345	1,83201721	1,83507249
7500	1,82862604	1,83255796	1,83628139
10000	1,82903096	1,83367269	1,83622355



RENDIMIENTO GFLOP/S (sin O2)			
N\P	25	50	100
100	0,53795371	0,34714604	0,34907718
1000	0,75599047	0,36565745	0,35887128
2500	0,75608595	0,36655172	0,35863395
5000	0,75644247	0,3506634	0,35125778
7500	0,75695999	0,3657899	0,35843093
10000	0,75623287	0,36661192	0,3588752



Práctica 2

Paralelización con OpenMP

Introducción

En la anterior práctica se realizó un programa secuencial que calculaba la Energía en cada punto y la Energía Acumulada de cada partícula, de una superficie NxN donde impactaban P partículas. En el documento de la anterior práctica se hizo un estudio sobre su coste temporal e implementación. En esta práctica se va a paralelizar dicho algoritmo con la librería OpenMP, con el fin de minimizar el coste temporal de ejecución. Al tratar con múltiples hilos en esta práctica se ha optado por usar la letra H para representar el número de hilos (se ha elegido una letra mayúscula ya que el resto de tallas también utilizan un letra mayúscula).

Dependencias de datos

Antes de comenzar a paralelizar, se ha realizado un análisis de dependencias con el fin de poder ver que operaciones dependen de que otras se hayan hecho previamente, y así evitar resultados incorrectos a la hora de paralelizar provocados por no respetar dicho orden.

En primer lugar la función A, que calcula el factor de atenuación no tiene ninguna dependencia de datos, no depende de la ejecución de ninguna otra función. Para calcular de distancia euclídea se pude calcular la diferencia de distancia en cada eje y elevar posteriormente al cuadrado por separado para cada uno de los ejes (x e y). Finalmente la raíz cuadrada y el exponencial dependen del sumatorio de la diferencia de cuadrados en cada eje, por tanto se debe de realizar en secuencial. El aporte de rendimiento que puede aportar calcular la diferencia de posición para cada uno de los ejes va a ser menor al tiempo de comunicación, es por ello que no vale la pena realizar una paralelización de esta sección de código.

En segundo lugar la función EA, que calcula la energía, su resultado es una expresión que primero se debe de calcular la función A para posteriormente mutiplicarse por N2 (que es calculado antes de llamar la función una única vez) y finalmente hacer la división de la energía de dicha partícula entre la expresión anterior. Por ello debe de ser secuencial.

En la función calc los cálculos principales. T se obtiene como resultado de sumar todos los valores de EA para una determinada partícula y E se obtiene tras sumar EA para todos los puntos del espacio. Por tanto primero es necesario obtener el resultado de EA para un punto y una partícula determinada y posteriormente sumarse al resultado determinado. Dichos sumatorios son se pueden paralelizar por reducción, pero anteriormente se deben de inicializar las variables antes. El cálculo de cada valor de EA es independiente de otros valores, por tanto el bucle es paralelizable y también todos los bloques de código (interiores y exteriores). Y además la carga de trabajo de todas la iteraciones es constante, cosa que es beneficiosa porque se podrá obtener mejor rendimiento con una planificación estática que tiene un menor coste de comunicación que las otras opciones de planificación.

Planteamiento de paralelización del cálculo (versión definitiva)

El cálculo principal consiste en 3 bucles (2 para recorrer todos los puntos de la superficie y 1 para recorrer todos los índices de la partículas), el orden de anidamiento para el cálculo secuencial es

irrelevante. Además es de tener en cuenta que todas las iteraciones del bucle tardan lo mismo, por tanto una planificación estática sería la idónea, el compilador asigna los procesos teniendo un coste de comunicación muy pequeño en tiempo de ejecución.

En el planteamiento paralelo se ha optado por poner los 2 bucles que recorren la superficie como exteriores y colapsarlos, obteniendo un bucle de N^2 iteraciones. Como el bucle exterior es el que se va a paralelizar conviene tener el mayor número de iteraciones, así los procesadores tienen tareas más pequeñas que se pueden dividir más equitativamente (teniendo de que todas las tallas de N son mayores que P).

El bucle interior recorre todas las partículas y es secuencial en cada hilo, no se ha optado por colapsar este bucle también porque para realizar las sumas para obtener T y P, en caso de optar por un triple colapso se tendría que haber atómicas ambas operaciones de suma para evitar la escritura simultanea múltiple (cosa que haría que el resto de hilos esperasen cuando se fuera a sumar) o se cada uno de los resultados de la función EA se fueran almacenando en una matriz tridimensional (sería muy eficiente teóricamente, pero a la hora de la experimentación se podría observar que reservar tanta memoria haría el acceso a las variables muy lento).

Dentro de los 3 bucles, cada elemento de E se puede calcular sin utilizar ningún mecanismo de sincronización de operaciones paralelas. Esto se debe a que cada una de las iteraciones de los 2 bucles colapsados de la superficie son independientes entre sí, por tanto solo accede un procesador al mismo valor de E[x][y].

No obstante, respecto al cálculo de T, si que hay accesos concurrentes hacía los mismos valores de T. Para ello se ha creado un array dinámico bidimensional llamado T_aux, cuyo índice de fila es la dimensión es el identificador del proceso y en la columna el índice de la partícula. Los punteros de la fila se inicializan de forma secuencial, pero los arrays dinámicos de cada columna, se crean reservando memoria local en cada hilo, de forma paralela. De la misma manera borramos cada una de las columnas en paralelo y borramos los punteros de cada fila en secuencial posteriormente. Cada resultado de EA se guarda en T_aux[tid][p] (siendo tid el identificador del proceso y p el índice de la particula). Para posteriormente sumar todos los elementos T_aux[tid][p] que poseean la misma p, posteriormente en otro bucle paralelo.

Para realizar la suma y obtener T a partir de T_aux. Es necesario 2 bucles (uno es para recorrer los índices de partículas y otro es para recorrer todos los identificadores de hilo). Se ha puesto como exterior y se ha paralelizado el acceso el número de partículas ya que P es mayor siempre al número de hilos y además no hace que se puedan sumar todos los resultados parciales sin usar reducción.

Cálculo del coste teórico en FLOP (versión definitiva)

Para el cálculo del coste teórico en FLOP, se ha extraído únicamente las partes del código en que se realizan operaciones con reales. Dichas operaciones están dentro de una zona paralela. Se ha decidido poder ver mejor como se ha realizado el cálculo. Los bucles parelelizados se muestran en azul, arriba de un conjunto de bucles anidados podemos se muestra cual es el coste de dicha zona de código.

Como N y P son tallas del problema, para representar el número de hilos se ha utilizado la letra H.

Se explicó en cuanto costaba cuanto costaba EA, en el apartado ***cálculo del coste secuencial.***

Para el cálculo de los **bucles paralelos** (con planificación estática q=1) se ha utilizado la siguiente formula:

$$\left\lceil \frac{\text{nº iteraciones}}{\text{nº hilos}} \right\rceil \cdot \text{operaciones interiores}$$

El dividir entre el número de hilos, redondear hacía arriba y posteriormente multiplicar por las operaciones interiores se debe a que es el coste del hilo que más tarda. La tarea es dividida entre tanto hilos como halla disponible, pero si no es posible hacer una división exacta una tarea se lleva un coste adicional. En caso de colapso se cuentan todas las iteraciones del bucle, como si fueran uno solo que contiene todas las iteraciones (como se puede observar en el primer bucle.

Como se esta usando planificación estática, como se ha mencionado anteriormente, no habrá coste a la hora de repartir la carga de trabajo a cada uno de los procesos a tiempo de ejecución, ya que esta planificación lo hace a tiempo de compilación. No obstante como es necesario instanciar y destruir una matiz T_aux de tamaño HxP y se un tiempo de sincronización al esperar que todos los hilos terminen el triple bucle anidados para calcular EA, para posteriormente realizar las reducción de T[p] , se tendrá en cuenta a la hora de contabilizar el coste contabilizado como tp(H,P).

Se ha utilizado un tamaño de chunk por defecto la hora de especificar que se va a usar la paralelización estática, es decir un tamaño de **nº de iteraciones/nº de hilos.** Esta planificación permite otorga iteraciones consecutivas a los procesos, permitiendo un acceso a memoria más eficiente, pero si el número de iteraciones no es divisible, todo el resto ira a parar al mismo procesador haciendo el resto de iteraciones.

Para los bucles no paralelos se ha utilizado la misma técnica para calcularlos que en el apartado en el apartado ***cálculo del coste secuencial.***

```
// ceil(N^2/H) * 25 P FLOP
#pragma omp for schedule(static) collapse(2) private(EA_pxy)
for (unsigned X=0; X<N; X++) {  // * N
    for (unsigned Y=0; Y<N; Y++) {  // * N
        for (unsigned p=0; p<P; p++) { // * P
            EA_pxy_ = EA(p,X,Y);          // +23 FLOP
            T_aux[tid][p] += EA_pxy_;     // +1  FLOP
            E[X][Y] += EA_pxy_;           // +1  FLOP
        }
    }
}

// ceil(P/H) FLOP
#pragma omp for schedule(static)
for (unsigned p=0; p<P; p++){    // paralelo * P
    for (unsigned th=0; th<nthreads; th++) { // * H
        T[p] += T_aux[th][p];                // +1 FLOP
    }
}
```

Coste de total / Coste del hilo que más tarda

$$T(N,P,H) = \lceil N^2 / H \rceil \cdot 25P + \lceil P / H \rceil \cdot H + tp(H,P)$$

Este es el análisis para el coste total, que corresponde al hilo que más tarda. Para obtener el coste de los hijos que menos tardan en caso que N o P no sean divisibles entre H, simplemente hay que cambiar el redondeo hacía arriba, por el hacia abajo, Ya que las tareas del resto, se lo dejan a los hilos que más tardar.

Coste del hilo que menos tarda

$$T(N,P,H) = \lfloor N^2 / H \rfloor \cdot 25P + \lfloor P / H \rfloor H + tp(H,P)$$

OMP MFLOP P=1000						
N\H	1	2	3	4	6	12
300	2250,001	2250,002	2250,003	2250,004	2250,006	2250,012
600	9000,001	9000,002	9000,003	9000,004	9000,006	9000,012
900	20250,001	20250,002	20250,003	20250,004	20250,006	20250,012
1200	36000,001	36000,002	36000,003	36000,004	36000,006	36000,012
1500	56250,001	56250,002	56250,003	56250,004	56250,006	56250,012
1800	81000,001	81000,002	81000,003	81000,004	81000,006	81000,012
2100	110250,001	110250,002	110250,003	110250,004	110250,006	110250,012

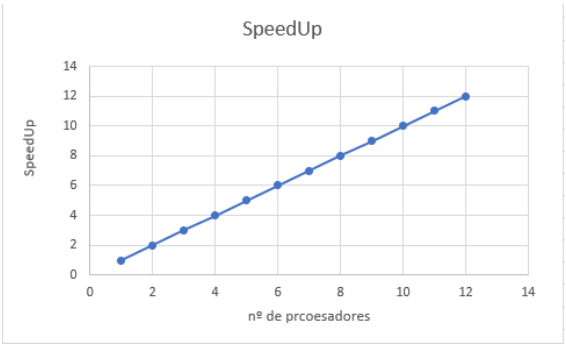
Estudio analítico del SpeedUp

El SpeedUp es un ratio que representa la velocidad ganada al paralelizar, es el resultado de dividir el tiempo secuencial entre el tiempo que tardaría con un hijo. Dichos tiempos ya se han calculado previamente, así que solo hay que dividirlos. Para este caso vamos a suponer que **cada FLOP tarda ta.**

$$S(H) = T(1) / T(P) = (25 \cdot N^2 \cdot P \cdot ta) / ((\lceil N^2 / H \rceil \cdot 25P + \lceil P / H \rceil \cdot H) \cdot ta + tp(H,P))$$

Dicha expresión puede ser algo compleja de entender, por tanto se va a mostrar en una gráfica para poder realizar un análisis sobre dicha función. Para ello se vamos a realizar las pruebas se le va a calcular los valores de H de 0 a 12 hilos con valores de P=50, N=5000, ta=1 y despreciando lo que vale tp (ya que no es posible determinar el valor de dicha variable, hasta que se realicen los experimentos).

H	SpeedUp
1	0,999999998
2	1,999999993
3	2,999999744
4	3,999999971
5	4,99999996
6	5,999998011
7	6,999997112
8	7,999999869
9	8,99999914
10	9,999999808
11	10,99999622
12	11,99999583



Como podemos observar, nuestro SpeedUp es lineal, teniendo un rendimiento teórico muy aproximado al 100% (casi perfecto). Eso significa que la velocidad aproximadamente directamente proporcional al número de núcleos. No obstante queda por computar determinar el coste de tp y por tanto puede que estos resultados no cuadren con la realidad).

A partir de aquí es todo nuevo

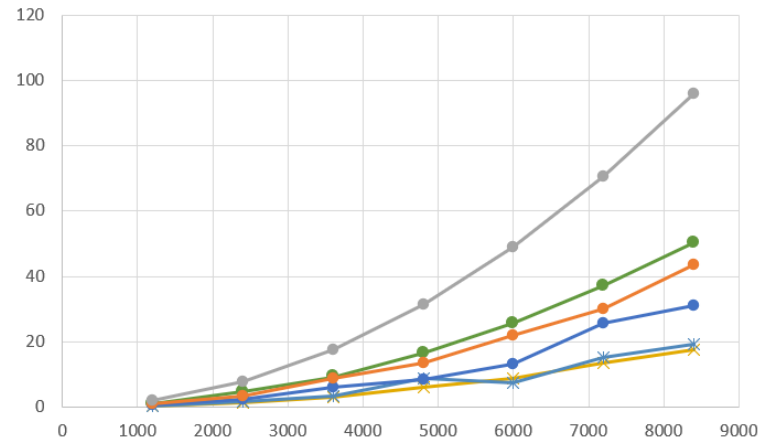
Estudio experimental del tiempo

Las tallas escogidas para el estudio experimentas son P=100 con N=1200 hasta 8400 en incrementos 1200 y P=1000 de N=300 hasta 2100 en incrementos de 300, con el fin de poder observar que pasa al paralelizar con tallas pequeñas y grandes de N, y como varía respecto a P . Se han realizado los experimentos con resultados secuenciales que tardan 1 minuto aproximadamente que para poder estudiar lo que se reduce el tiempo al paralelizar.

Los estudios experimentales se han realizado con compute-0-3 del clúster que proporciona la universidad. Dicha máquina posee 6 núcleos físico y 12 lógicos. Para los test se ha compilado el programa utilizando la opción de compilación -O2. Los resultados mostrados están en segundos.

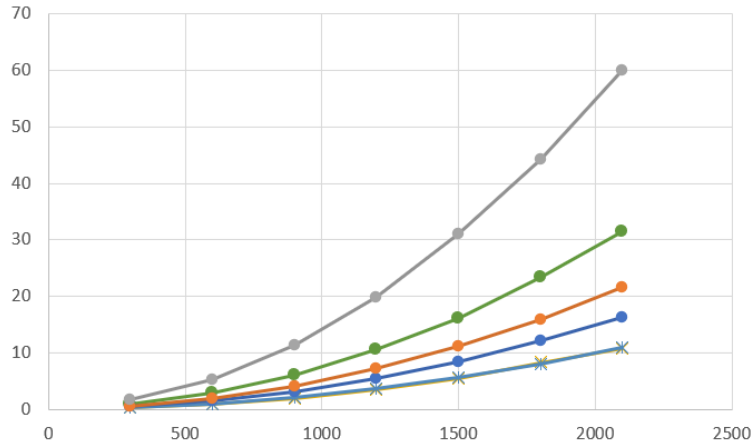
Con P=100 podemos observar que se reduce el tiempo de ejecución considerablemente al pasar de 1 hilo a 2, pero lastimosamente parece ganar menos tiempo con 3 y 4 hilos, y los resultado resultan menos precisos a partir de la talla P=5000 con más de 3 hilos. También se han usado tallas N divisibles entre el número de hilos con el objetivo de que todos los hilos tengan la misma carga de trabajo y facilitar el análisis, al no preocuparse de que con determinada talla hay hilos que deben de realizar un mayor número de iteraciones.

OMP P=100						
N\H	1	2	3	4	6	12
1200	2,11	1,12	0,98	0,55	0,32	0,36
2400	7,87	4,57	3,35	2,44	1,58	1,46
3600	17,67	9,27	8,73	6,15	3,22	3,08
4800	31,38	16,46	13,39	8,51	8,88	5,91
6000	49	25,7	21,99	13,29	7,36	8,87
7200	70,56	37,03	30	25,72	15,04	13,51
8400	96,06	50,39	43,7	31,12	19,21	17,51



Al aumentar la talla P a 1000, se ha decidió reducir las tallas a medir con el fin de reducir los tiempos y poder sacar una gráfica que describa mejor lo que sucede con tallas N más pequeñas, pero con mayor carga por proceso. Se puede observar que los resultados están más relacionados con el modelo teórico previamente descrito, siendo el número de hilos inversamente proporcional al tiempo.

OMP P=1000						
N\H	1	2	3	4	6	12
300	1,61	0,84	0,6	0,45	0,31	0,26
600	5,2	2,8	1,94	1,43	0,96	0,92
900	11,36	5,96	4,12	3,14	2,06	1,91
1200	19,86	10,53	7,25	5,38	3,62	3,5
1500	31,13	16,15	11,17	8,34	5,7	5,36
1800	44,16	23,4	15,96	12,09	8,04	8,3
2100	59,99	31,44	21,64	16,25	10,94	10,78



En ambas gráficas podemos observar que con 6 y 12 hilos, los tiempos son similares (no se llegan a distinguir en la última gráfica las dos curvas están muy juntas). Esto se debe a que el número de núcleos de la máquina es 6 y al poner 12, al mismo núcleo le tocan 2 hilos en vez de 1. Posteriormente en el resto de apartados se realizará un análisis más profundo.

Estudio experimental de la aceleración y la eficiencia.

La acelaración (SpeedUp) es un valor que representa cuatas veces es paralelización de rápida respecto a una implementación secuencial, es por ello que se calcula como:

tiempoSecuencial/tiempoParalelo.

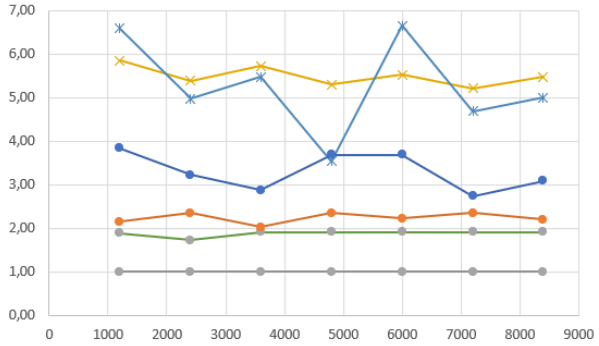
La eficiencia es mide cuanta velocidad se ha ganado por cada hilo que se ha utilizado y se obtiene mediante la siguente operación:

speedUp/nº de hilos

Dichos valores son realmente útiles para medir la calidad de una paralelización.

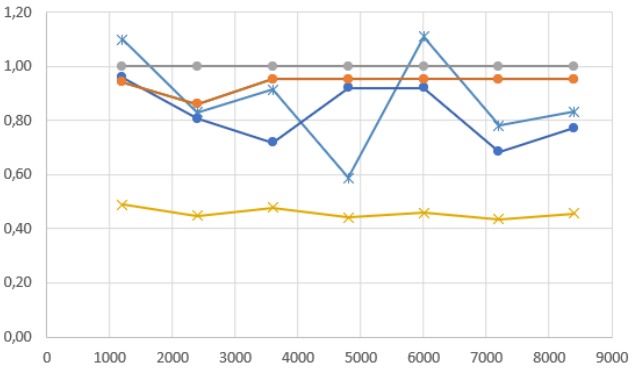
SpeedUp

OMP SpeedUp P=100						
N\H	1	2	3	4	6	12
1200	1,00	1,88	2,15	3,84	6,59	5,86
2400	1,00	1,72	2,35	3,23	4,98	5,39
3600	1,00	1,91	2,02	2,87	5,49	5,74
4800	1,00	1,91	2,34	3,69	3,53	5,31
6000	1,00	1,91	2,23	3,69	6,66	5,52
7200	1,00	1,91	2,35	2,74	4,69	5,22
8400	1,00	1,91	2,20	3,09	5,00	5,49



Eficiencia

OMP Eficiencia P=100						
N\H	1	2	3	4	6	12
1200	1,00	0,94	0,72	0,96	1,10	0,49
2400	1,00	0,86	0,78	0,81	0,83	0,45
3600	1,00	0,95	0,67	0,72	0,91	0,48
4800	1,00	0,95	0,78	0,92	0,59	0,44
6000	1,00	0,95	0,74	0,92	1,11	0,46
7200	1,00	0,95	0,78	0,69	0,78	0,44
8400	1,00	0,95	0,73	0,77	0,83	0,46

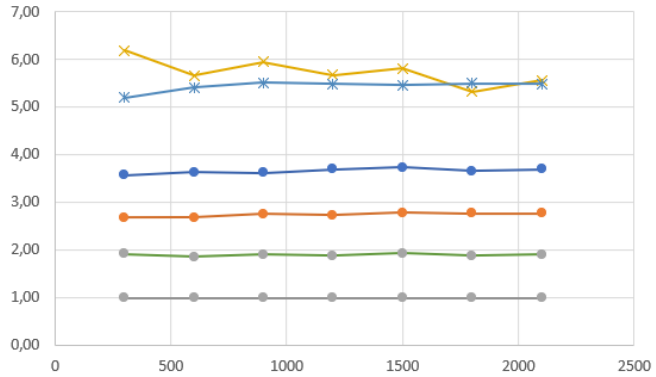


El speedUp debería de ser de siempre menor o igual al número de hilos que se esta empleando y por consecuencia, la eficiencia menor o igual a uno. No obstante se puede observar que hay datos en H=6 que superan la eficiencia de 1, por algún motivo, tras repetir varias veces el experimento. Es posible que en esta en prueba en algún momento haya interferido otro usuario a la hora de medir del experimento, o algún otro factor que desconozco.

No obstante se puede observar que aumentando la eficiencia va disminuyendo hasta llegar a 6 hilos y posteriormente en 12 baja a la mitad. Con P=1000 se puede observar con más precisión.

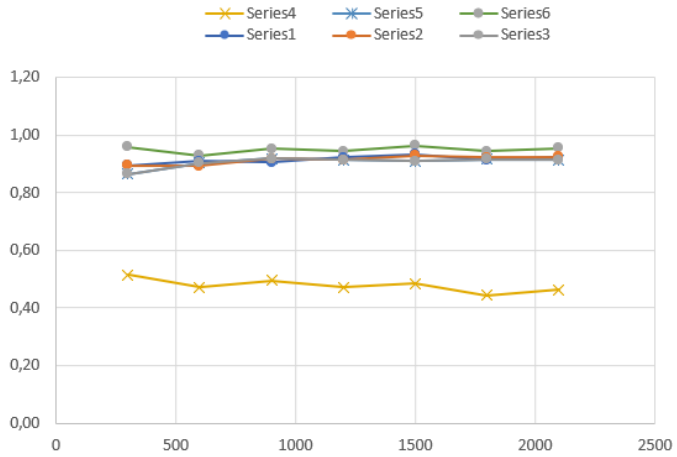
SpeedUp

OMP SpeedUp P=1000						
N\H	1	2	3	4	6	12
300	1,00	1,92	2,68	3,58	5,19	6,19
600	1,00	1,86	2,68	3,64	5,42	5,65
900	1,00	1,91	2,76	3,62	5,51	5,95
1200	1,00	1,89	2,74	3,69	5,49	5,67
1500	1,00	1,93	2,79	3,73	5,46	5,81
1800	1,00	1,89	2,77	3,65	5,49	5,32
2100	1,00	1,91	2,77	3,69	5,48	5,56



Eficiencia

OMP Eficiencia P=1000						
N\H	1	2	3	4	6	12
300	1,00	0,96	0,89	0,89	0,87	0,52
600	1,00	0,93	0,89	0,91	0,90	0,47
900	1,00	0,95	0,92	0,90	0,92	0,50
1200	1,00	0,94	0,91	0,92	0,91	0,47
1500	1,00	0,96	0,93	0,93	0,91	0,48
1800	1,00	0,94	0,92	0,91	0,92	0,44
2100	1,00	0,95	0,92	0,92	0,91	0,46



Con los experimentos con P=1000 los resultados obtenidos han sido realmente buenos, muy cercanos a una paralelización perfecta (eficiencia=1), pero se ha alcanzado dicho valor, ya que es un ideal que no se puede obtener debido al tiempo de sincronización que existe entre for del cálculo y el for de la acomulación. Se puede observar que conforme se aumenta el número de hilos decrementa un poco la velocidad que se gana por hilo, pero aún así sigue siendo muy rentable incrementar hasta 6 hilos ganando un 90% de velocidad por hilo.

También se observa que con tallas muy pequeñas de N la paralelización pierde efectividad pasando del 87% con tallas pequeñas al 91% con H=6, presentando una diferencia del 4%. Con H=4 y H=3, también hay presente una diferencia pero va menguando.

Otra observación es que hay un dato (H=12 N=300) donde el ratio en tanto por uno del speedUp obtenido es mayor que el número de procesadores físicos. Esto puede haber sido provocado porque en el caso de H=12, el algoritmo puede que haya hecho uso de los núcleos lógicos de la CPU.

Cálculo de los MFLOPS teóricos

Para el cálculo de la velocidad de procesamiento teórica es necesario calcular el número de operaciones de coma flotante que realiza cada unidad de tiempo.

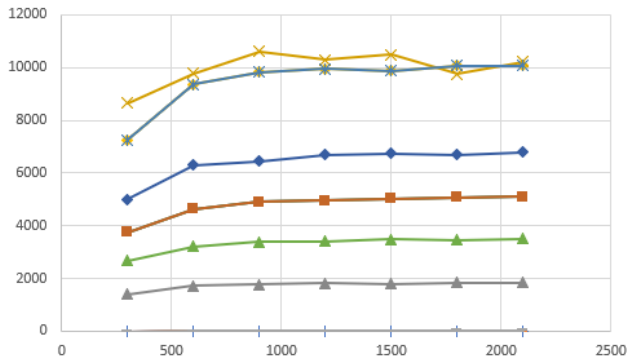
Se ha optado por primero calcular el número de operaciones que se realizan en total en el programa entre todos los núcleos, **25*P*N^2 + H*P** (siendo el primer sumando el número las operaciones de el primer bucle donde se calculan todos los valores el segundo sumando es el segundo bucle que hace todas las acumulaciones del vector T). Para obtener el resultado en MFLOP se ha dividido **la expresión anterior entre un millón** dando como resultado las siguientes tablas.

OMP MFLOP P=1000						
N\H	1	2	3	4	6	12
300	2250,001	2250,002	2250,003	2250,004	2250,006	2250,012
600	9000,001	9000,002	9000,003	9000,004	9000,006	9000,012
900	20250,001	20250,002	20250,003	20250,004	20250,006	20250,012
1200	36000,001	36000,002	36000,003	36000,004	36000,006	36000,012
1500	56250,001	56250,002	56250,003	56250,004	56250,006	56250,012
1800	81000,001	81000,002	81000,003	81000,004	81000,006	81000,012
2100	110250,001	110250,002	110250,003	110250,004	110250,006	110250,012

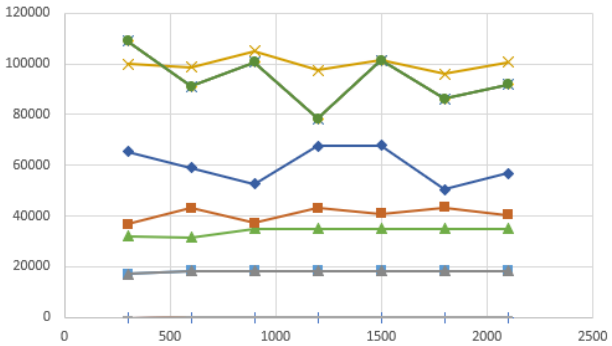
OMP MFLOP P=100						
N\H	1	2	3	4	6	12
1200	36000,001	36000,002	36000,003	36000,004	36000,006	36000,012
2400	144000,001	144000,002	144000,003	144000,004	144000,006	144000,012
3600	324000,001	324000,002	324000,003	324000,004	324000,006	324000,012
4800	576000,001	576000,002	576000,003	576000,004	576000,006	576000,012
6000	900000,001	900000,002	900000,003	900000,004	900000,006	900000,012
7200	1296000,001	1296000,002	1296000,003	1296000,004	1296000,006	1296000,012
8400	1764000,001	1764000,002	1764000,003	1764000,004	1764000,006	1764000,012

Posteriormente se divide entre el tiempo en segundos que hemos obtenido en los resultados experimentales, y así obtenemos la velocidad de procesamiento en MFLOPs.

OMP MFLOPS P=1000						
N\H	1	2	3	4	6	12
300	1397,516149	2678,57381	3750,005	5000,008889	7258,083871	8653,892308
600	1730,769423	3214,286429	4639,176804	6293,709091	9375,00625	9782,621739
900	1782,570511	3397,651342	4915,049272	6449,04586	9830,1	10602,10052
1200	1812,688872	3418,803609	4965,517655	6691,450558	9944,753039	10285,71771
1500	1806,938677	3482,97226	5035,810474	6744,604796	9868,422105	10494,40522
1800	1834,239153	3461,538547	5075,188158	6699,752192	10074,62761	9759,03759
2100	1837,806318	3506,679453	5094,732116	6784,615631	10077,69707	10227,27384



OMP MFLOPS P=100						
N\H	1	2	3	4	6	12
300	17061,61185	32142,85893	36734,69694	65454,55273	109090,9273	100000,0333
600	18297,33177	31509,84726	42985,07552	59016,39508	91139,2443	98630,14521
900	18336,16304	34951,45653	37113,40241	52682,92748	100621,1199	105194,8091
1200	18355,64057	34993,92479	43017,17722	67685,07685	78260,87038	97461,93096
1500	18367,34696	35019,45533	40927,69454	67720,09059	101351,352	101465,6158
1800	18367,34695	34998,6498	43200,0001	50388,80264	86170,21316	95928,94241
2100	18363,52281	35006,94586	40366,13279	56683,80476	91827,17366	100742,4336



Cada núcleo de la CPU tiene 2100MHz de frecuencia (teniendo en cuenta que se esta usando la máquina compute-0-3 del clúster y el dato ha sido **cat /proc/cpuinfo.** Igual que pasa con el secuencial en paralelo también se haya el problema que con tallas muy pequeñas de N la carga de trabajo no es lo suficientemente grande para utilizar toda la potencia del procesador para el cálculo de operaciones reales, no obstante al aumentar el tamaño de N esto deja de ser un problema.

En el ejemplo de arriba donde P=1000 se puede observar como todos los núcleos no llegan a su velocidad pico hasta llegar a un talla de N=700 aproximadamente, también dependiendo del número de hilos, a mayor número de hilos más le costará llegar al tope.

También se puede observar que al aumentar el número de procesadores aumenta la capacidad de calculo por segundo del procesador, pero cada vez se decrementa más como se comentó en el apartado del speedup y la acelaración (pero no se ve a simple vista ya que el algoritmo tiene una eficiencia bastante buena).

Los capacidad de cálculo con 12 núcleos es similar a la de 6 hilos, ya que le procesador como se ha comentado antes solo tiene 6 núcleos físicos y 12 lógicos. Pero aún así ha habido una pequeña mejoría debido a estos hilos lógicos

Estudio de otras implementaciones.

Se ha probado a variar las diversas planificaciones con las que el programa hace el reparto del bucle paralelo que calcula los diversos resultados de EA. Para estos experimentos se han usado las mismas tallas que en el apartado anterior, con el número de hilos (H=6).

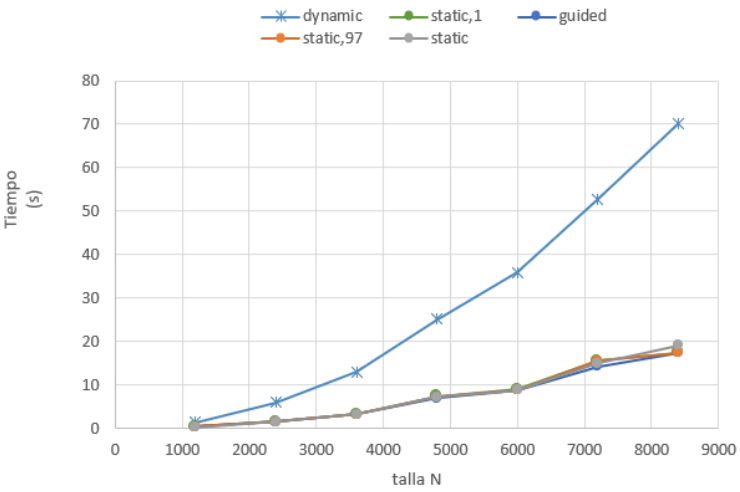
En primer lugar se ha probado en cambiar el tamaño de chunck de la máquina a 1 y 97. Con q=1 las tareas se reparten de 1 en 1 y la tarea que la iteración que más tarda tiene solo una iteración más, pero los procesos tienen partes de memoria que no son continuas haciendo que el acceso a la memoria sea un poco más lento en algunos casos, pero tampoco supone una gran diferencia.

En cambio con el chunk fujado a 97, las asignaciones de memoria son de 97 a 97, creando secciones de memoria continua más largas para cada iteración, pero puede haber una iteración que recorra hasta 96 iteraciones más. Lo que pasa es que es un número de iteraciones tan pequeño que no suele suponer una gran carga computacional dicho extra.

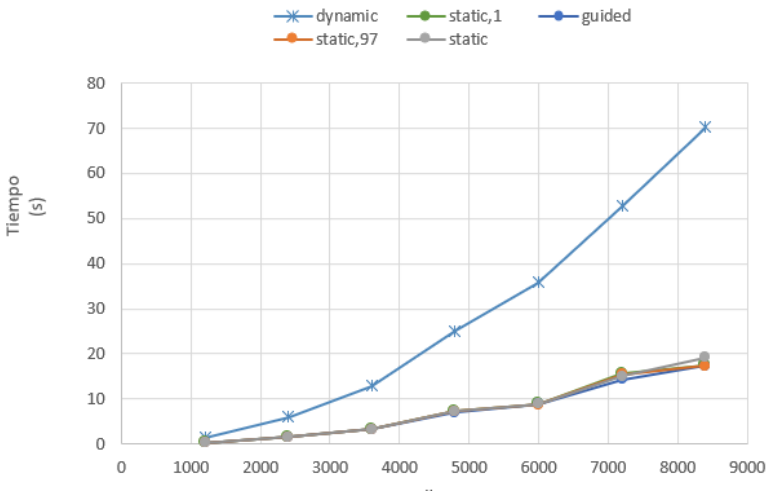
Con dynamic la asignación de las iteraciones a los procesadores se realiza a tiempo de ejecución. Por defecto usa un tamaño de chunk de 1, por tanto implica de que hace el reparto más equitativo posible, pero a cambio cada vez que realiza una iteración hay un tiempo de comunicación extra. Previamente se comentó de que no era buena idea usar este tipo de planificación ya que como todas las iteraciones duran lo mismo un reparto a tiempo de compilación sería la optimo. Los resultados experimentales, determinan que la hipótesis era verídica tardando 3,5 veces más que el resto de planificaciones.

Finamente, la planificación guided es una planificación dinámica en el que el tamaño del chunk comienza siendo más grande con el fin de minimizar el tiempo de comunicaciones y posteriormente va haciéndose más pequeño para poder realizar un mejor ajuste. No suele ser una planificación recomendada en casos en que se sepa que todas la iteraciones tienen la misma carga de trabajo, pero aún así en este tipo de casos debería obtener mejores resultados que dynamic. No obstante los resultados han sido muy similares a los de una planificación estática.

OMP P=100					
N	static	static,1	static,97	guided	dynamic
1200	0,32	0,37	0,37	0,37	1,44
2400	1,58	1,6	1,59	1,57	6,03
3600	3,22	3,22	3,2	3,2	12,91
4800	7,36	7,37	7,3	7	25,12
6000	8,88	8,9	8,8	8,86	35,85
7200	15,04	15,56	15,48	14,31	52,81
8400	19,21	17,5	17,4	17,4	70,3



OMP P=1000					
N	static	static,1	static,97	guided	dynamic
300	0,31	0,3	0,3	0,29	0,93
600	0,96	0,96	0,95	0,94	3,51
900	2,06	2,06	2,05	2,05	7,8
1200	3,62	3,61	3,59	3,61	13,39
1500	5,7	5,71	5,77	5,56	23,88
1800	8,04	8,01	8,05	7,97	31
2100	10,94	10,92	10,92	10,86	42,13


dynamic
static,1
guided
static,97
static
Tiempo (s)
80
70
60
50
40
30
20
10
0
0
1000
2000
3000
4000
5000
6000
7000
8000
9000

Práctica 3

Paralelización en MPI

Introducción

En esta práctica el objetivo es paralelizar en mismo programa secuencial de la simulación de particulas utilizando la librería de paso de mensajes MPI, en vez de usar Open MP. MPI nos va a permitir utilizar varios nodos compute del clúster con el fin de hacer más rápida la ejecución del programa.

A la hora de hacer un algoritmo paralelo siempre es recomendable hacer un análisis de las dependencias, antes de planear cual va a ser la estrategia que emplear en función de las limitaciones que nos impone las dependencias de datos. No obstante, al tratarse del mismo algoritmo a paralelizar las dependencias van a ser las mismas que en el aparatado de **Dependencia de Datos OMP** (por tanto si aún no lo ha visto, puede ser de gran interés verlo antes de leer todo el resto de la práctica).

Una cosa a tener en cuenta es que al tratarse de una simulación, solo el primer nodo **(el master rank=0)** es el único proceso que posee los datos de entrada N, P, posición y energía de cada partícula, ya que a la hora de hacer un experimento real solo un proceso sería el encargado de recoger todos los datos de los impactos de las partículas sobre la superficie. Es importante que ya que el proceso 0 pueda compartir los datos necesarios al resto de procesos para que puedan realizar las paralelizaciones.

También es importante tener la talla a paralelizar debe de ser divisible entre el número de procesos, con el fin de facilitar la implementación.

Estrategia de paralelización

Estrategia subóptima

En esta estrategia el objetivo es paralelizar el bucle que recorre el índice de las partículas. Para ello primero primero es necesario obtener P

Estrategia óptima

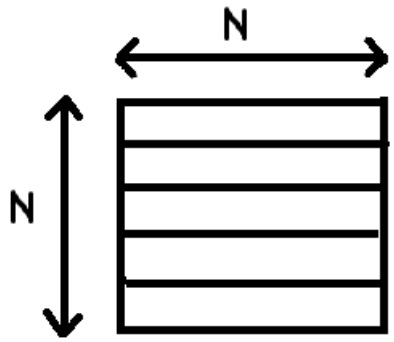
En esta estrategia el objetivo es paralelizar el bucle que recorre la superficie, dividiendo la superficie en trozos en trozos equitativos. Es mejor que paralelizar el número de partículas ya que no hace falta hacer la reducción para obtener todos los valores de la matriz EA.

En primer lugar se hace se obtiene el identificador de cada procesos con Comm_rank y el identificador número de procesos con Comm_size para obtener el número de procesos totales.

En primer logar compartimos con Bcast los valores de N y P. Una vez sabemos las tallas del problema le compartimos a todos los procesos con otros Bcast todos los valores de la posición en x y en y de cada partícula y su energía, ya que hemos dividido la superficie y no las partículas cualquier superficie, esto supone un coste extra respecto la paralelización del bucle de particulas.

Se reserva un array local a cada proceso llamado lT de longitud P, que guarda los resultados parciales de cada proceso para posteriormente aplicar un Reduce a todos los elementos y obtener T.

También se reserva una matriz de lE de memoria continua de tamaño **N/nºprocesos x N**, que guardar una parte de los resultados finales de la matiz E. Para reservar la memoria continua se a creado un array llamado lE_mem que contiene **N^2/ nºprocesos** (dicho array se utilizará también más adelante). Para obtener la matriz lE se ha hecho que cada uno de los filas apunten a la dirección de memoria donde se supone que empieza su primera columna. La superficie E es repartida de la siguiente manera:



Una vez generadas las estructuras de datos anteriores se procede al cálculo de los valores de EA, usando un tiple bucle anidado, en esta paralelización el bucle exterior que recorre la posición en x de la placa en cada proceso recorre desde **X=0 mientras X<N/size**, pero la partícula como el índice es local y se opera con el global es necesario hacer una conversión para obtener este a la hora de calcular la distancia. Dicha conversión es **globalX = localX+rank*(N/size)**. El resultado de EA se guarda en el arrays temporales lT y TE.

Para obtener todos los resultados se reducen los P elementos del array T.

Finalmente se hace un Gather todas los resultados de E locales juntando todos los resultado de procesos guardados en el array lE_mem.

Estudio analítico del coste del programa

Coste de comunicaciones

Antes de empezar esta vez el nº de hilos de ejecución del programa, se va a representar con la palabra size porque en MPI se obtiene dicho parámetro usando el método MPI_Comm_Size y en la práctica anterior el uso de la letra mayúscula resulto poco intuitivo.

Para dicho análisis tendremos en cuenta:

- m: nº de elementos a enviar o recibir en un mensaje
- t_{in}: Tiempo en incialización
- t_b: Tiempo que tarda en transmitir un byte.

Al tratar con elementos de tipo float e int, t_b va a ser de 4 en ambos casos.

En primer lugar se hacen 3 MPI_B_cast, para compartir los 3 arrays de P elementos.

```
MPI_Bcast (x, P, MPI_INT,
           0, MPI_COMM_WORLD);

MPI_Bcast (y, P, MPI_INT,
            0, MPI_COMM_WORLD);

MPI_Bcast (e, P, MPI_FLOAT,
```

```
            0, MPI_COMM_WORLD);
```

El coste de cada uno de los Bcast es de **lg size *(t_{in} + m * t_b)**. En todos ellos **m = 4 * P** debido a que el tamaño de un float y de un un int es 4.

Después de realizar las operaciones aritméticas locales a cada proceso, se realiza un un Gather, juntar los datos locales a cada proceso E_mem en matriz de memoria continua E del proceso 0.

```
MPI_Gather (lE_mem, lN2, MPI_FLOAT,
            rank==0? E[0]:NULL, lN2, MPI_FLOAT,
            0, MPI_COMM_WORLD);
```

Dicha operación tiene un coste de **lg size * t_{in} + m * (size-1)/size * t_b.** En este caso **m = 4 * N^2**, ya que ln guarda es N^2/size (localN^2).

Finalmente se hace un Reduce de todos los elementos de cada uno de los elementos del array T de cada uno de los procesos, y se almacena en el proceso 0.

```
MPI_Reduce(lT, T , P, MPI_FLOAT, MPI_SUM, 0, MPI_COMM_WORLD);
```

Dicha operación tiene un coste aritmético que se tratará posteriormente, pero lo veremos en el siguiente subapartado, ya que este se dedica exclusivamente al coste de comunicaciones. El coste de las comunicaciones de la función es **lg size *(t_{in} + m * t_b)**, igual que en el Bcast, pero en esta llamada **m = 4 * P**.

Sumando todos los costes calculados anteriormente tenemos un coste total de operaciones de:

3 lg size *(t_{in} + 4P* t_b) + lg size * t_{in} + 4N^2 * (size-1)/size * t_b + lg size *(t_{in} + 4P * t_b)

Si hacemos una simplificación del resultado obtenido obtenemos:

lg size * (6 t_{in} + 16P*t_b) + lg size * t_{in} + 4N^2

Coste computacional

```
for (unsigned X=0; X<lN; X++) // * N/size
  for (unsigned Y=0; Y<N; Y++) // * N
     for(unsigned p=0; p<P; p++){ // * P
        float dx = ((int) X+rank*lN) - ((int)x[p]);
        dx *= dx;  // + 1 FLOP

        float dy = ((int) Y) - ((int)y[p]);
        dy *= dy;  // + 1 FLOP
        float A_pxy_ = exp(sqrt(dx+dy)); // + 8+8+1 FLOP

        float EA_pxy_ = e[p] / ( A_pxy_*N2 ); // + 4+1 FLOP

        lT[p] += EA_pxy_;    // + 1 FLOP
        lE[X][Y] += EA_pxy_; // + 1 FLOP
      }
```

$$T(N,P,size) = \sum_{p=0}^{N/size-1} \sum_{x=0}^{N-1} \sum_{y=0}^{N-1} (25) = 25 \cdot \frac{\mathrm{N}}{\mathrm{size}} \cdot \mathrm{N} \cdot \mathrm{P} = 25 \cdot \frac{\mathrm{N}^2}{size} \cdot \mathrm{P}$$

El resultado mostrado anteriormente, sería el coste de calcular el bucle anterior. No obstante todo no hemos tenido en cuenta que se realiza una reducción.

```
MPI_Reduce(lT, T , P, MPI_FLOAT, MPI_SUM, 0, MPI_COMM_WORLD);
```

El coste de dicha operación se computa como **lg size * op**, siendo op el coste en FLOP de la operación, en este caso al ser una suma solo cuesta 1 FLOP.

Finalmente nos queda un coste computacional por núcleo, teniendo en cuenta que el reduce se hace de forma conjunta:

$$T(N,P,size) = 25 \cdot \frac{\mathrm{N}^2}{size}\mathrm{P} + \lg size \ \ \mathrm{P}$$

No obstante, el coste computacional de en FLOP de todas las operaciones que realiza el programa es:

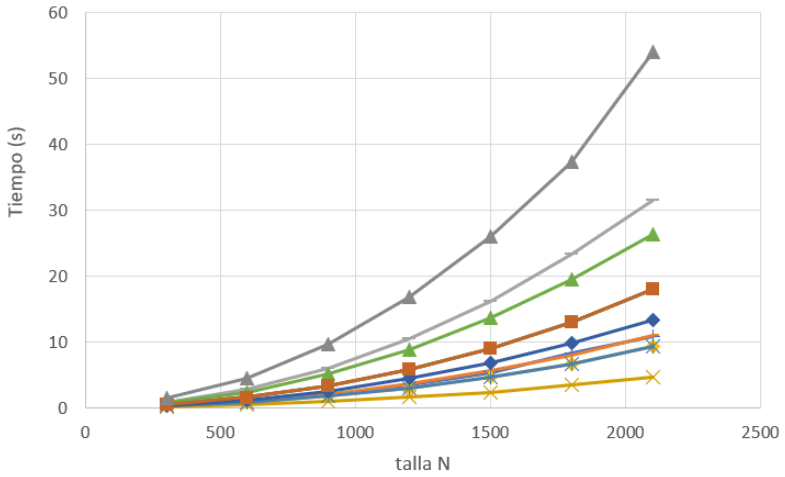
$$T(N,P,size) = 25\mathrm{N}^2\mathrm{P} + \lg size \ \ \mathrm{P}$$

Resultados experimentales

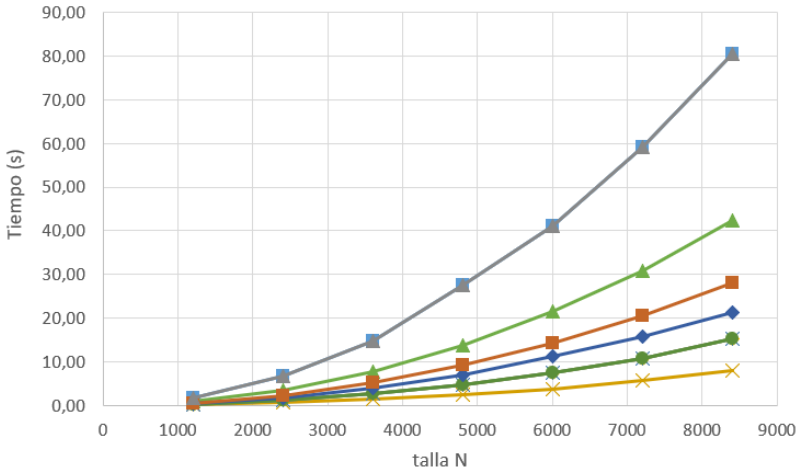
Para los experimentos se ha utilizado el ordenador boe que proporciona la universidad de valencia. Dicho ordenador posee 12 núcleos físicos y 24 núcleos físicos, con una frecuencia en todos los núcleos de 2’1 GHz (Intel(R) Xeon(R) CPU E5-2620 v2 @ 2.10GHz). Dicha información ha sido obtenida con el comando cat /proc/cpuinfo.

Se ha probado con las diversas tallas de N para P=100 y P=1000, dicha son las misma que se utilizaron en la práctica de OpenMP, así nos permite comparar la paralelización realizada. Las tallas de N son múltiplos del número de procesadores para evitar resultados erróneos. Para compilar el programa se ha utilizado la opción de optimización del compilador -O2. El tiempo mostrado en las siguientes tablas y gráficas esta en segundos.

Tiempo (s) MPI P=1000						
N\size	1	2	3	4	6	12
300	1,41	0,75	0,51	0,39	0,26	0,17
600	4,55	2,39	1,57	1,23	0,83	0,54
900	9,65	5,12	3,36	2,55	1,75	0,96
1200	16,83	8,79	5,86	4,42	3,04	1,59
1500	25,92	13,62	9,00	6,83	4,73	2,38
1800	37,32	19,46	12,97	9,82	6,72	3,54
2100	54,08	26,28	17,92	13,28	9,31	4,58



Tiempo(s) MPI P=100						
N\size	1	2	3	4	6	12
1200	1,69	0,89	0,59	0,46	0,31	0,18
2400	6,64	3,47	2,31	1,75	1,20	0,62
3600	14,86	7,79	5,19	3,92	2,69	1,39
4800	27,56	13,71	9,20	6,99	4,79	2,57
6000	41,13	21,61	14,33	11,31	7,45	3,75
7200	59,17	30,80	20,53	15,69	10,72	5,74
8400	80,38	42,27	28,12	21,38	15,16	7,98



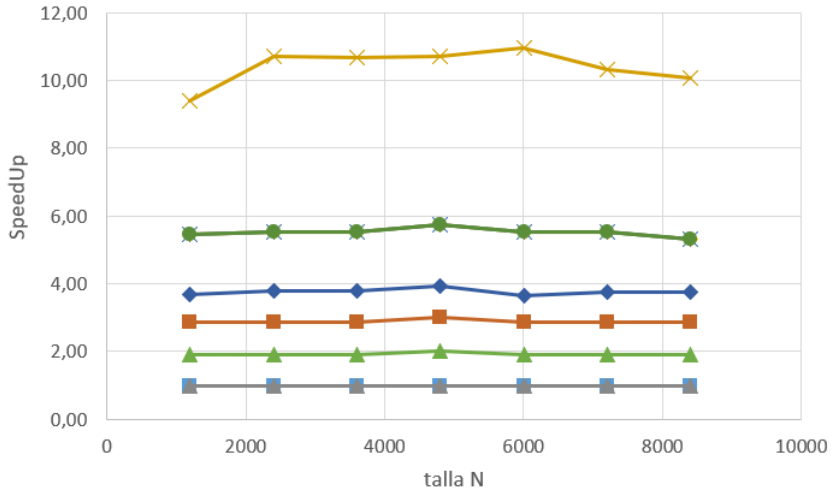
Como podemos observar al realizar la prueba con un sistema que si que tiene 12 núcleos físicos si que podemos observar una mejoría notable con 12 hilos, a diferencia de las pruebas que realizamos con compute-0-3 en la práctica de OpenMP.

Se puede observar de que el número de núcleos es aproximadamente inversamente proporcional al tiempo que tarda el programa en ejecutarse. Al doblar el número de núcleos se reduce a la mitad el tiempo de ejecución, esto es un resultado de una buena paralelización y de una alta eficiencia, no obstante eso se verá en más profundidad en posteriores apartados.

SpeedUp del algoritmo

Para el cálculo del speed se divide el tiempo en paralelo entre el tiempo secuencial. Es un ratio que cuantas veces es más rápido un programa respecto a su versión secuencial.

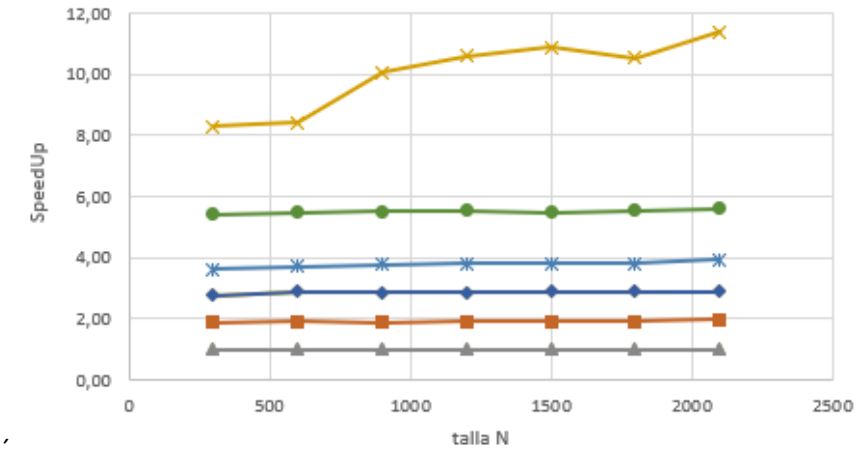
SpeedUp MPI P=100						
N\size	1	2	3	4	6	12
1200	1,00	1,90	2,86	3,67	5,45	9,39
2400	1,00	1,91	2,87	3,79	5,53	10,71
3600	1,00	1,91	2,86	3,79	5,52	10,69
4800	1,00	2,00	3,00	3,94	5,75	10,72
6000	1,00	1,90	2,87	3,64	5,52	10,97
7200	1,00	1,92	2,88	3,77	5,52	10,31
8400	1,00	1,90	2,86	3,76	5,30	10,07



En este caso parece ganar al añadir tras añadir se gana aproximadamente 90% de velocidad, esto se ve muy bien reflejado con 1 hasta 6 núcleos (observa los resultados de la talla N=3600, donde se observa con mejor claridad). Con 12 núcleos el la ganancia de rendimiento es similar pero tiene es mucho más variable.

El SpeedUp se mantiene constante conforme aumenta la talla de N, presentando una muy ligera bajada cuando N es muy grande (aproximadamente N>5000) y aumenta el número de hilos debido al coste del Gather de la matriz E. Dicha perdida de rendimiento es dependiente del número de hilos tanto como del tamaño de N.

SpeedUp MPI P=1000						
N\size	1	2	3	4	6	12
300	1,00	1,88	2,76	3,62	5,42	8,29
600	1,00	1,90	2,90	3,70	5,48	8,43
900	1,00	1,88	2,87	3,78	5,51	10,05
1200	1,00	1,91	2,87	3,81	5,54	10,58
1500	1,00	1,90	2,88	3,80	5,48	10,89
1800	1,00	1,92	2,88	3,80	5,55	10,54
2100	1,00	1,98	2,91	3,92	5,59	11,37

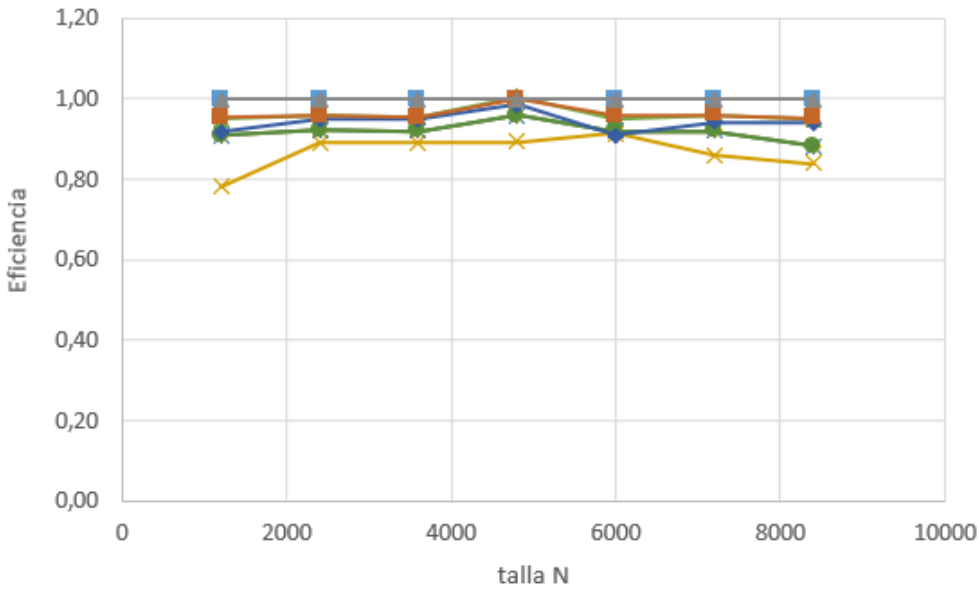


Con tallas más pequeñas de N, podemos observar que gana mucha menos velocidad con tallas más pequeñas de N, con size=12 siendo solo 8 veces más rápido que su versión secuencial con N=300. Pero a partir de tallas como N=900, el algoritmo pasa a ser 10 veces más rápido, y con N=2100 el algoritmo llega a ser 11,37 veces más rápido.

Eficiencia del algoritmo

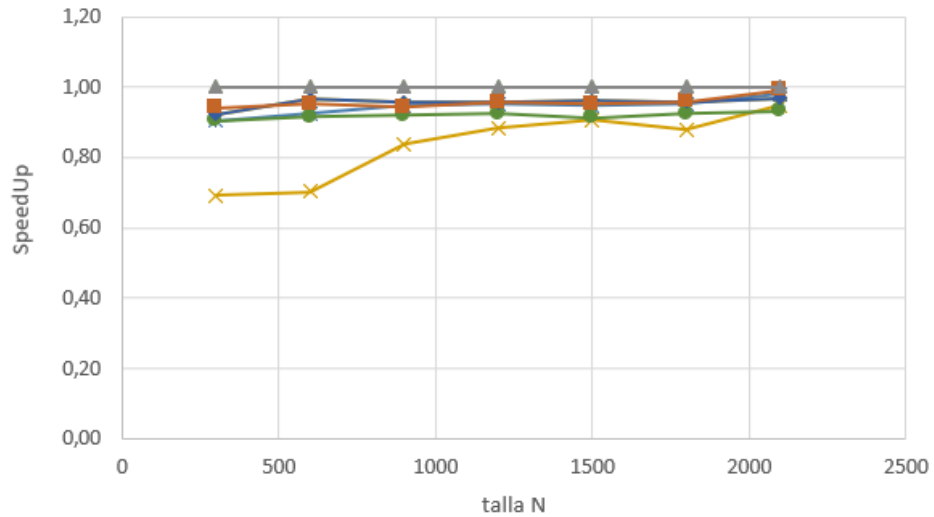
La eficiencia es la cantidad de velocidad de procesamiento ganada por núcleo añadido al sistema o al algoritmo, para calcularla se ha divido el SpeedUp y se ha dividido entre el número de núcleos. Esta medida puede ser útil para medir la rentabilidad de añadir núcleos al sistema, y medir la calidad del paralelismo.

Eficiencia MPI P=100						
N\size	1	2	3	4	6	12
1200	1,00	0,95	0,95	0,92	0,91	0,78
2400	1,00	0,96	0,96	0,95	0,92	0,89
3600	1,00	0,95	0,95	0,95	0,92	0,89
4800	1,00	1,00	1,00	0,99	0,96	0,89
6000	1,00	0,95	0,96	0,91	0,92	0,91
7200	1,00	0,96	0,96	0,94	0,92	0,86
8400	1,00	0,95	0,95	0,94	0,88	0,84



Como podemos observar el algoritmo tiene una eficiencia muy buena, que va disminuyendo muy poco a poco conforme vamos aumentando la cantidad de núcleos, teniendo una eficiencia de alrededor de 95% con 2 núcleos y alrededor de 90% con 6 núcleos . Aquí se ve más claro que aproximadamente que la ejecución es un aproximadamente un 90% más rápida al ir añadiendo núcleos (de 1 a 6 núcleos).

Eficiencia MPI P=100						
N\size	1	2	3	4	6	12
300	1,00	0,94	0,92	0,90	0,90	0,69
600	1,00	0,95	0,97	0,92	0,91	0,70
900	1,00	0,94	0,96	0,95	0,92	0,84
1200	1,00	0,96	0,96	0,95	0,92	0,88
1500	1,00	0,95	0,96	0,95	0,91	0,91
1800	1,00	0,96	0,96	0,95	0,93	0,88
2100	1,00	0,99	0,97	0,98	0,93	0,95



Al disminuir la talla de P, la eficiencia con mayor número de procesadores disminuye en menor medida. Esto se debe a que al hacer un 3 BCast de vectores P elementos al principio y al final un

Reduce para sumar los elementos de cada proceso del array T que posee P elementos también, el coste de comunicaciones es muy dependiente de dicha talla.

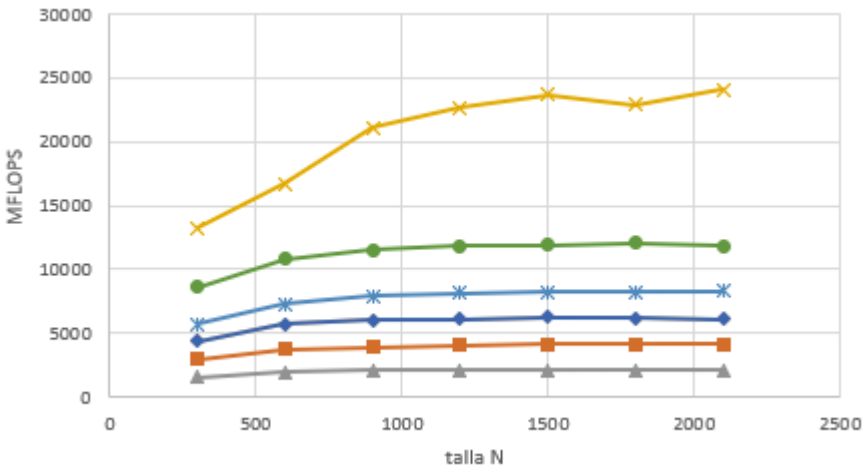
MFLOPS

El número de operaciones de coma flotante por segundos se ha calculado usando la formula anterior del coste computacional para todos los hilos de ejecución del programa. Para obtener el resultado en MFLOP, se ha dividido el resultado entre un millón. Posteriormente para calcular los MFLOPS se divide el resultado anterior ente el tiempo de ejecución del programa.

Los MFLOPS son millones de operaciones de coma flotante por segundo un sistema computacional. Teniendo en cuenta en cuenta que el ordenador BOE posee 12 núcleos a 2'1 GHz de frecuencia base y 2'5GHz de frecuencia máxima.

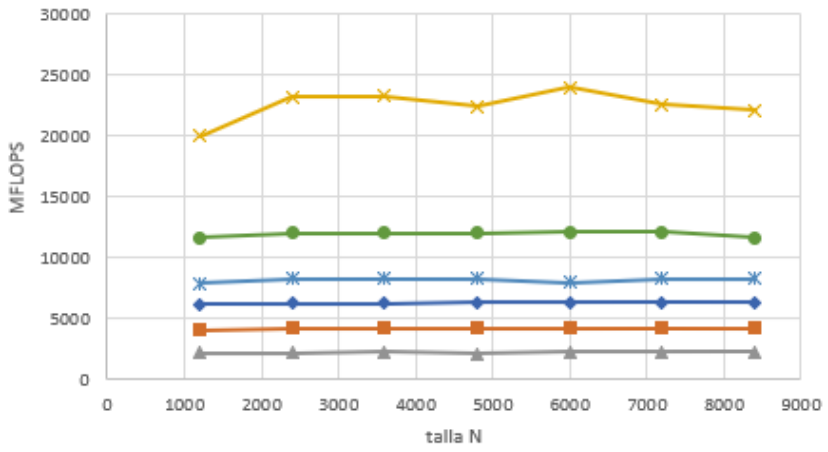
MFLOP MPI P=1000						
N\H	1	2	3	4	6	12
300	225	225	225	225	225	225
600	900	900	900	900	900	900
900	2025	2025	2025	2025	2025	2025
1200	3600	3600	3600	3600	3600	3600
1500	5625	5625	5625	5625	5625	5625
1800	8100	8100	8100	8100	8100	8100
2100	11025	11025	11025	11025	11025	11025

MFLOPS MPI P=1000						
N\H	1	2	3	4	6	12
300	1596	3000	4412	5769	8654	13235
600	1978	3766	5732	7317	10843	16667
900	2098	3955	6027	7941	11571	21094
1200	2139	4096	6143	8145	11842	22642
1500	2170	4130	6250	8236	11892	23634
1800	2170	4162	6245	8248	12054	22881
2100	2117	4195	6152	8302	11842	24072



MFLOPS MPI P=100						
N\H	1	2	3	4	6	12
1200	3600	3600	3600	3600	3600	3600
2400	14400	14400	14400	14400	14400	14400
3600	32400	32400	32400	32400	32400	32400
4800	57600	57600	57600	57600	57600	57600
6000	90000	90000	90000	90000	90000	90000
7200	129600	129600	129600	129600	129600	129600
8400	176400	176400	176400	176400	176400	176400

MFLOPS MPI P=100						
N\H	1	2	3	4	6	12
1200	2130	4045	6102	7826	11613	20000
2400	2169	4150	6234	8229	12000	23226
3600	2180	4159	6243	8265	12045	23309
4800	2090	4201	6261	8240	12025	22412
6000	2188	4165	6281	7958	12081	24000
7200	2190	4208	6313	8260	12090	22578
8400	2195	4173	6273	8251	11636	22105



Como se puede observar para que un núcleo llegue a utilizar todo el rendimiento del procesador hace falta que la talla de N sea mayor o igual a 500, con P=1000, no obstante con 12 núcleos sería necesario una talla mucho mayor como se puede observar en la tabla.

En cambio con P=100, las tallas con un solo núcleo se han llegado a 2100 MFLOPS con todas las tallas de N, que es el número aproximado que operaciones de coma flotante que se permite hacer con una frecuencia de 2’1GHz.

En ambos casos al duplicar el número de procesadores se multiplica obtiene aproximadamente un poco menos del doble de velocidad. Como hemos visto en estudios anteriores.

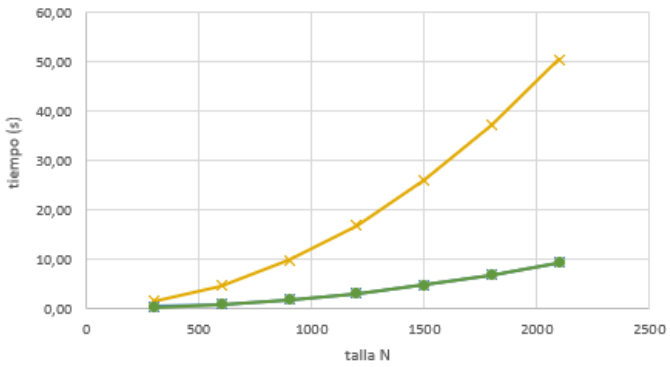
Estudio de otras opciones de paralelización

Se va medir la diferencia entre paralelizar utilizando los mismos núcleos de la CPU y utilizar núcleos de distintos ordenadores. Para ello he hecho una prueba con donde se pone se ha ejecutado el comando se ha utilizado un fichero host que incluye el propio ordenador boe con 5 computes. Se han realizado varias veces los experimentos para confirmar que el resultado no ha haya sido influenciado por otros sucesos no previstos (sistema 1).

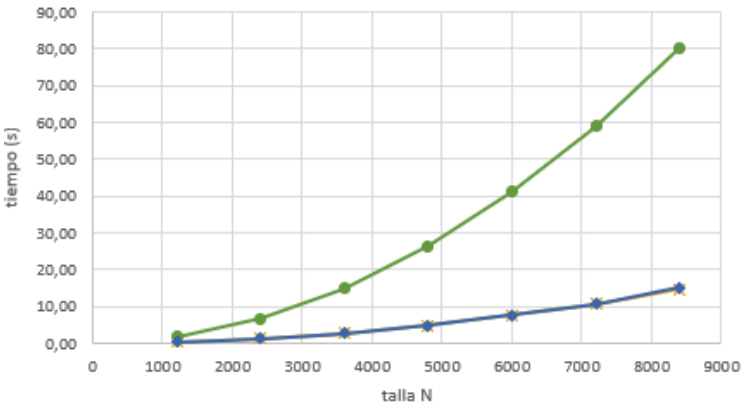
También se ha probado con 6 nucleos en 2 ordenadores compute, con la opción -np 6. Para ver si mejora el tiempo respecto a 6 núcleos (sistema 2).

El sistema 3, es la máquina boe con 6 núcleos, con la que se va realizar la comparación.

Tiempo (otras opciones) MPI P=1000			
N\Sistema	1	2	3
300	1,42	0,27	0,26
600	4,56	0,82	0,83
900	9,68	1,76	1,75
1200	16,82	3,06	3,04
1500	26,03	4,80	4,73
1800	37,10	6,71	6,72
2100	50,37	9,34	9,31



Tiempo (otras opciones) MPI P=100			
N\H	1	2	3
1200	0,37	1,69	0,31
2400	1,22	6,61	1,20
3600	2,70	14,81	2,69
4800	4,80	26,29	4,79
6000	7,46	41,13	7,45
7200	10,71	59,01	10,72
8400	14,57	80,33	15,16



Como se puede observar utilizando 6 ordenadores con 1 núcleo por ordenador no se obtiene una mejora sustancial respecto a la ejecución secuencial debido al alto coste de comunicación que supone comunicar varias máquinas.

Con 2 ordenadores con 6 procesos cada uno se obtienen resultados ligeramente peores (imperceptibles en las gráficas) que los obtenidos con un ordenador con 6 procesos. Si aumentamos la complejidad de los cálculos y redujéramos la cantidad de mensajes que se intercambian entre computadores, podríamos obtener mejores resultados usando varios ordenadores, pero este no ha sido el caso.

Anexo1: Código en C++ secuencial

```
/**
 * @author Santiago Millán Giner
 * @file secuencial.cpp
 */

#include <iostream>
#include <algorithm>
#include <math.h>
#include <iomanip>
#include <time.h>
using namespace std;

// VARABLES QUE DETERMINAN LA TALLA DEL PROBLEMA //
unsigned N;     // ancho/alto de la superficie cuadrada
unsigned P;     // número de particulas de muestra

// ARRAYS DINÁMICOS DE ENTRADA DE DATOS DE PARTICULAS //
unsigned* x;    // posiciones de las particulas en eje X
unsigned* y;    // posiciones de las particulas en eje Y
float* e;       // energía de las particulas (siempre positivo)

// ARRAYS DINÁMICOS DE RESULTADOS //
float*  T;      // soluciones de T para todas las particulas
float** E;      // soluciones de E para todas las posiciones x∈[0..N[
y∈[0..N[

// VARIABLES AUXILIARES //
unsigned N2;    // variable auxiliar que guarda N² para evitar
calcularla varias veces

/**
 * Calcula y devuelve A(p,x,y)
 * @param p Índice de la particula
 * @param X Posición de el punto en el eje X sobre el cual se quiere
calcular la distancia
 * @param Y Posición de el punto en el eje Y sobre el cual se quiere
calcular la distancia
 * @return Factor de atenuación
 */
float A(const unsigned p, const unsigned X, const unsigned Y) {
    // El cast a entero de unsigned evita el underflow
    float dx = ((int) X) - ((int)x[p]);    dx *= dx;  // dx = (x-xp)²
    float dy = ((int) Y) - ((int)y[p]);    dy *= dy;  // dy = (y-yp)²
    return exp(sqrt(dx+dy));
}

/**
 * Calcula y devuelve EA(p,x,y)
```

```
 * @param p Índice de la particula
 * @param X Posición de X del puto sobre que se quiere calcular la
distancia
 * @param Y Posición de Y del puto sobre que se quiere calcular la
distancia
 * @return Energía acumlada de la particula p respecto al punto (x,y)
 */
float EA(const unsigned p, const unsigned X, const unsigned Y) {
    return e[p] / ( A(p,X,Y)*N2 );
}

/**
 * Calcula T(p) para todas las particulas p
 * Calcula E(x,y) para todas la superficie NxN
 */
void calc() {
    N2 = N*N;

    float EA_pxy_; //< guarda el resultado de EA de forma temporal
para un valor de p,x,y determinado

    for (unsigned X=0; X<N; X++)
        for (unsigned Y=0; Y<N; Y++)
            E[X][Y] = 0;

    for (unsigned p=0; p<P; p++) {
        T[p] = 0;
        for (unsigned X=0; X<N; X++)
            for (unsigned Y=0; Y<N; Y++) {
                EA_pxy_ = EA(p,X,Y);
                T[p] += EA_pxy_;
                E[X][Y] += EA_pxy_;
            }
    }
}

/**
 * Inicializa todos los arrays dinámicos globales que existen en el
programa
 */
void init_arrays() {
    x = new unsigned [P];
    y = new unsigned [P];
    e = new float [P];

    T = new float [P];

    // los resultados de E se guardan en una matriz NxN de posiciones
de memoria consecutivas
```

```
    float* E_mem = (float *) malloc (sizeof(float) * N * N); //
Reserva N² posiciones de floats
    E = (float **) malloc (sizeof(float*) * N);              //
Reserva N arrarys de float
    for (int i=0; i<N; i++)
        E[i] = &(E_mem[i*N]); // Cada posición de E[i] apunta a la
primera posición de cada fila de N elementos
}

/**
 * (Código del enunciado)
 * Obtiene el instante de tiempo actual en segundos
 * @return Instnte de tiempo actual en segundos
 */
double get_time() {
    struct timespec t;
    clock_gettime(CLOCK_REALTIME, &t);
    return (double) t.tv_sec + ((double) t.tv_nsec)/1e9;
}

/**
 * Realiza una simulación con valores de energía de 100 a 101000
 * Y posiciones dentro de la superficie [0..N-1] x [0..N-1]
 */
void experimentacion() {
    cout << "MODO EXPERIMENTACION" << endl;
    cout << "Introduce los siguientes valores" << endl;
    cout << "P = "; cin >> P;
    cout << "N = "; cin >> N;

    init_arrays();

    for (unsigned p=0; p<0; p++) {
        x[p] = (unsigned) ((N - 1) * (rand() / (float) RAND_MAX) +
0.5);
        y[p] = (unsigned) ((N - 1) * (rand() / (float) RAND_MAX) +
0.5);
        e[p] = 100 + 10000 * (rand() / (float) RAND_MAX);
    }

    double t0 = get_time();
    calc();
    double t1 = get_time();

    cout << "Tiempo de ejecución = " << t1 - t0 << "s" << endl;
}

/**
 * Función con la finalidad para comprobar que el procedimiento de
caluclo es correcto.
```

```
 *
 * Calcula T(p) para todas las particulas p
 * y E(x,y) para toda la superficie NxN
 * para la siguente muestra de datos:
 *
 * (x1 , y1 , e1 ) = (0, 0, 100)
 * (x2 , y2 , e2 ) = (2, 0, 200)
 * (x3 , y3 , e3 ) = (3, 2, 50)
 *
 * Posteiromente se muestra por pantalla.
 */
void depuracion() {
    cout << "MODO DEPURACION" << endl;

    N = 4;
    P = 3;

    init_arrays();

    x[0]=0;     y[0]=0;     e[0]=100.0;
    x[1]=2;     y[1]=0;     e[1]=200.0;
    x[2]=3;     y[2]=2;     e[2]= 50.0;

    calc();

    cout << "T(p)" << endl;
    for (unsigned p=0; p<P; p++)
        cout << "T(" << p << ") = " << T[p] << endl;

    cout<< endl;

    cout << "E(x,y)" << endl;
    for (unsigned i=0; i<N; i++) {
        for (unsigned j=0; j<N; j++)
            cout << fixed << setprecision(2) <<E[j][i] << '\t';
        cout << endl;
    }
}

/**
 * Función principal, selección del modo de ejecución
 * @return devuelve 0 si el programa finaliza con éxito
 */
int main() {
    cout << "Introduzca 0 para el MODO DEPURACION" << endl;
    cout << "Introduzca 1 para el MODO EXPEREMENTACION" << endl;

    // modo_exp indica si esta en modo experimentación
    bool modo_exp;
    cin >> modo_exp;
```

```
    if (modo_exp)
        experimentacion();
    else
        depuracion();

    return 0;
}
```

Anexo 2: Paralelización con OMP

```
/**
 * @author Santiago Millán Giner
 * @file secuencial.cpp
 */

#include <vector>
#include <omp.h>
#include <iostream>
#include <algorithm>
#include <math.h>
#include <iomanip>
#include <time.h>
using namespace std;

// VARABLES QUE DETERMINAN LA TALLA DEL PROBLEMA //
unsigned N;     // ancho/alto de la superficie cuadrada
unsigned P;     // número de particulas de muestra

// ARRAYS DINÁMICOS DE ENTRADA DE DATOS DE PARTICULAS //
unsigned* x;    // posiciones de las particulas en eje X
unsigned* y;    // posiciones de las particulas en eje Y
float* e;       // energía de las particulas (siempre positivo)

// ARRAYS DINÁMICOS DE RESULTADOS //
float*  T;      // soluciones de T para todas las particulas
float** E;      // soluciones de E para todas las posiciones x∈[0..N[
y∈[0..N[

// VARIABLES AUXILIARES //
unsigned N2;    // variable auxiliar que guarda N² para evitar
calcularla varias veces

/**
 * Calcula y devuelve A(p,x,y)
 * @param p Índice de la particula
 * @param X Posición de el punto en el eje X sobre el cual se quiere
calcular la distancia
 * @param Y Posición de el punto en el eje Y sobre el cual se quiere
calcular la distancia
 * @return Factor de atenuación
 */
float A(const unsigned p, const unsigned X, const unsigned Y) {
    // El cast a entero de unsigned evita el underflow
    float dx = ((int) X) - ((int)x[p]);    dx *= dx;  // dx = (x-xp)²
    float dy = ((int) Y) - ((int)y[p]);    dy *= dy;  // dy = (y-yp)²
    return exp(sqrt(dx+dy));
}
```

```
/**
 * Calcula y devuelve EA(p,x,y)
 * @param p Índice de la particula
 * @param X Posición de X del puto sobre que se quiere calcular la
distancia
 * @param Y Posición de Y del puto sobre que se quiere calcular la
distancia
 * @return Energía acumlada de la particula p respecto al punto (x,y)
 */
float EA(const unsigned p, const unsigned X, const unsigned Y) {
    return e[p] / ( A(p,X,Y)*N2 );
}

/**
 * Calcula T(p) para todas las particulas p
 * Calcula E(x,y) para todas la superficie NxN
 */
void calc() {
    N2 = N*N;

    float EA_pxy_; //< guarda el resultado de EA de forma temporal
para un valor de p,x,y determinado

    for (unsigned X=0; X<N; X++)
        for (unsigned Y=0; Y<N; Y++)
            E[X][Y] = 0;

    for (unsigned p=0; p<P; p++) {
        float Tp = 0.0;
    }

    // T Auxiliar es una matiz de floats de nthreads x P
    // en ella se almacenan los resultados parciales de T[p] con cada
hilo, para sumarse y obtener el final
    float** T_aux;

    #pragma omp parallel
    {
        unsigned nthreads = omp_get_num_threads(); // Número de hilos
        int tid = omp_get_thread_num();            // Identidicador
del hilo

        #pragma omp single
        {
            //cout << "nthreads = " << nthreads << endl;

            T_aux = new float* [nthreads];
            /*
            for(int i=0; i<nthreads; i++) {
```

```
            T_aux[i] = new float[P];
            for (unsigned p=0; p<P; p++)
                T_aux[i][p] = 0;
        }
        */
};

T_aux[tid] = new float[P];
for (unsigned p=0; p<P; p++)
    T_aux[tid][p] = 0;

/*
// Mostrar hilos después de la inicialización
#pragma omp critical
{
    cout << "Hola soy el hilo " << tid << endl;
    for (unsigned p=0; p<P; p++)
        cout << T_aux[tid][p] << endl;
    cout << endl;
};
*/

#pragma omp for schedule(static) collapse(2) private(EA_pxy)
for (unsigned X=0; X<N; X++) {  // paralelo
    for (unsigned Y=0; Y<N; Y++) {  // paralelo
        for (unsigned p=0; p<P; p++) {
            EA_pxy_ = EA(p,X,Y);
            T_aux[tid][p] += EA_pxy_;
            E[X][Y] += EA_pxy_;
        }
    }
}

/*
// Mostrar resultados parciles
#pragma omp critical
{
    cout << "Adios soy el hilo " << tid << endl;
    for(int p=0; p<P; p++)
        cout <<T_aux[tid][p] <<endl;
    cout<<endl;
};
 */

#pragma omp for schedule(static)
for (unsigned p=0; p<P; p++){
    for (unsigned th=0; th<nthreads; th++) {
        T[p] += T_aux[th][p];
    }
}
```

```
        // Borrar T_aux
        delete[] T_aux[tid];
        #pragma omp single
        {
            delete [] T_aux;
        };
    }
}

/**
 * Inicializa todos los arrays dinámicos globales que existen en el
programa
 */
void init_arrays() {
    x = new unsigned [P];
    y = new unsigned [P];
    e = new float [P];

    T = new float [P];

    // los resultados de E se guardan en una matriz NxN de posiciones
de memoria consecutivas
    float* E_mem = (float *) malloc (sizeof(float) * N * N); //
Reserva N² posiciones de floats
    E = (float **) malloc (sizeof(float*) * N);              //
Reserva N arrarys de float
    for (int i=0; i<N; i++)
        E[i] = &(E_mem[i*N]); // Cada posición de E[i] apunta a la
primera posición de cada fila de N elementos
}

/**
 * (Código del enunciado)
 * Obtiene el instante de tiempo actual en segundos
 * @return Instnte de tiempo actual en segundos
 */
double get_time() {
    struct timespec t;
    clock_gettime(CLOCK_REALTIME, &t);
    return (double) t.tv_sec + ((double) t.tv_nsec)/1e9;
}

/**
 * Realiza una simulación con valores de energía de 100 a 101000
 * Y posiciones dentro de la superficie [0..N-1] x [0..N-1]
 * ¡ Recuerda que N y P deben de estar inicializadas !
 */
void experimentacion() {
```

```
    init_arrays();

    for (unsigned p=0; p<P; p++) {
        x[p] = (unsigned) ((N - 1) * (rand() / (float) RAND_MAX) +
0.5);
        y[p] = (unsigned) ((N - 1) * (rand() / (float) RAND_MAX) +
0.5);
        e[p] = 100 + 10000 * (rand() / (float) RAND_MAX);
    }

    double t0 = get_time();
    calc();
    double t1 = get_time();

    cout << "Tiempo de ejecución = " << t1 - t0 << "s" << endl;

    // Borra los arrays dinámicos, para que no influyan en otros
experimentos
    delete[] x;
    delete[] y;
    delete[] e;

    delete[] T;
    delete[] E;
}

/**
 * Función con la finalidad para comprobar que el procedimiento de
caluclo es correcto.
 *
 * Calcula T(p) para todas las particulas p
 * y E(x,y) para toda la superficie NxN
 * para la siguente muestra de datos:
 *
 * (x1 , y1 , e1 ) = (0, 0, 100)
 * (x2 , y2 , e2 ) = (2, 0, 200)
 * (x3 , y3 , e3 ) = (3, 2, 50)
 *
 * Posteiromente se muestra por pantalla.
 */
void depuracion() {
    N = 4;
    P = 3;

    init_arrays();

    x[0]=0;     y[0]=0;     e[0]=100.0;
    x[1]=2;     y[1]=0;     e[1]=200.0;
    x[2]=3;     y[2]=2;     e[2]= 50.0;
```

```
    calc();

    cout << "T(p)" << endl;
    for (unsigned p=0; p<P; p++)
        cout << "T(" << p << ") = " << T[p] << endl;

    cout<< endl;

    cout << "E(x,y)" << endl;
    for (unsigned i=0; i<N; i++) {
        for (unsigned j=0; j<N; j++)
            cout << fixed << setprecision(2) <<E[j][i] << '\t';
        cout << endl;
    }
}

/**
 * Realiza mulitples experimentos, con las tallas de las tablas
 */
void multiple() {
    cout << "MODO MULTIPLE" << endl;

    vector<unsigned> tallasP ({25,50,100});
    vector<unsigned> tallasN ({100,1000, 2500, 5000, 7500, 10000});

    for(auto tallaP :tallasP) {
        for(auto tallaN: tallasN) {
            P = tallaP;
            N = tallaN;
            cout <<"P = "<< P;
            cout <<"N = "<< N;
            experimentacion();
        }
    }
}

/**
 * Función principal.
 * Introduce un argumento al comando para ejecutarlo en modo multiple.
 * Si no se pone dicha opción te deja elegir entre el modo depuraacion
o experimentación.
 * NOTA:
 *   El modo multiple esta pensado para ser ejecutado con un script,
mientras que el normal esta pensado para ser
 *   ejecutado manulmente.
 * @return devuelve 0 si el programa finaliza con éxito
 */
int main(int argc, char* argv[]) {
    if (argc > 1) {
        multiple();
```

```
    } else {
        cout << "Introudce 0 para el MODO DEPURACION" << endl;
        cout << "Intorduce 1 para el MODO EXPERIMENTACION" << endl;
        cout << "> ";

        // modo_exp indica si esta en modo experimentación
        bool modo_exp;
        cin >> modo_exp;

        if (modo_exp) {
            cout << "MODO EXPERIMENTACION" << endl;
            cout << "Introduce los siguientes valores" << endl;
            cout << "P = "; cin >> P;
            cout << "N = "; cin >> N;
            experimentacion();
        }
        else {
            cout << "MODO DEPURACION" << endl;
            depuracion();
        }
    }

    return 0;
}
```

Anexo 3: Implementación en OMP

```
/**
 * Implementación de la simulación en MPI basada en la paralelización
de OpenMP
 * @author Santiago Millán Giner
 * @file secuencial.cpp
 */
#include <mpi.h>
#include <iostream>
#include <algorithm>
#include <math.h>
#include <iomanip>
#include <time.h>
#include <cstdlib>
using namespace std;

// VARABLES QUE DETERMINAN LA TALLA DEL PROBLEMA //
unsigned N;     // ancho/alto de la superficie cuadrada
unsigned P;     // número de particulas de muestra

// ARRAYS DINÁMICOS DE ENTRADA DE DATOS DE PARTICULAS //
unsigned* x;    // posiciones de las particulas en eje X
unsigned* y;    // posiciones de las particulas en eje Y
float* e;       // energía de las particulas (siempre positivo)

// ARRAYS DINÁMICOS DE RESULTADOS //
float*  T;      // soluciones de T para todas las particulas
float** E;      // soluciones de E para todas las posiciones x∈[0..N[
y∈[0..N[

// VARIABLES AUXILIARES //
unsigned N2;    // variable auxiliar que guarda N² para evitar
calcularla varias veces

/**
 * Calcula y devuelve A(p,x,y)
 * @param p Índice de la particula
 * @param X Posición de el punto en el eje X sobre el cual se quiere
calcular la distancia
 * @param Y Posición de el punto en el eje Y sobre el cual se quiere
calcular la distancia
 * @return Factor de atenuación
 */
float A(const unsigned p, const unsigned X, const unsigned Y) {
    // El cast a entero de unsigned evita el underflow
    float dx = ((int) X) - ((int)x[p]);    dx *= dx;  // dx = (x-xp)²
    float dy = ((int) Y) - ((int)y[p]);    dy *= dy;  // dy = (y-yp)²
    return exp(sqrt(dx+dy));
}
```

```
/**
 * Calcula y devuelve EA(p,x,y)
 * @param p Índice de la particula
 * @param X Posición de X del puto sobre que se quiere calcular la
distancia
 * @param Y Posición de Y del puto sobre que se quiere calcular la
distancia
 * @return Energía acumlada de la particula p respecto al punto (x,y)
 */
float EA(const unsigned p, const unsigned X, const unsigned Y) {
    return e[p] / ( A(p,X,Y)*N2 );
}

/**
 * Inicializa todos los arrays dinámicos globales que existen en el
programa
 */
void init_arrays() {
    x = new unsigned [P];
    y = new unsigned [P];
    e = new float [P];

    T = new float [P];

    // los resultados de E se guardan en una matriz NxN de posiciones
de memoria consecutivas
    float* E_mem = (float *) malloc (sizeof(float) * N * N); //
Reserva N² posiciones de floats
    E = (float **) malloc (sizeof(float*) * N);              //
Reserva N arrarys de float
    for (int i=0; i<N; i++)
        E[i] = &(E_mem[i*N]); // Cada posición de E[i] apunta a la
primera posición de cada fila de N elementos

      // Instaciar todos los sumatorios a 0
      for(int i=0; i<N; i++)
            for(int j=0; j<N; j++)
                  E[i][j] = 0;

      for(int p=0; p<P; p++)
            T[p] = 0;
}

/**
 * (Código del enunciado)
 * Obtiene el instante de tiempo actual en segundos
 * @return Instnte de tiempo actual en segundos
 */
double get_time() {
```

```
    struct timespec t;
    clock_gettime(CLOCK_REALTIME, &t);
    return (double) t.tv_sec + ((double) t.tv_nsec)/1e9;
}


/**
 * Si se le pasan 2 el primero instancia N y el segundo P
 * en caso contrario, ejecuta los datos de ejemplo
 * @return devuelve 0 si el programa finaliza con éxito
 */
int main(int argc, char *argv[]) {
    MPI_Init(&argc, &argv);

      int rank, size;
      MPI_Comm_rank(MPI_COMM_WORLD, &rank);
      MPI_Comm_size(MPI_COMM_WORLD, &size);

      double t0, t1;

      // CARGAR DATOS DE ENTRADA
      if(rank==0) {
            if (argc>=3) {
                  N = atoi(argv[1]);
                  P = atoi(argv[2]);

                  init_arrays();

                  for (unsigned p=0; p<P; p++) {
                        x[p] = (unsigned) ((N - 1) * (rand() / (float)
RAND_MAX) + 0.5);
                        y[p] = (unsigned) ((N - 1) * (rand() / (float)
RAND_MAX) + 0.5);
                        e[p] = 100 + 10000 * (rand() / (float)
RAND_MAX);
                  }
            }
            else {
                  cout << "MODO DEPURACION" << endl;

                  N = 4;
                  P = 3;

                  init_arrays();

                  x[0]=0;     y[0]=0;     e[0]=100.0;
                  x[1]=2;     y[1]=0;     e[1]=200.0;
                  x[2]=3;     y[2]=2;     e[2]= 50.0;
            }
```

```
		t0 = get_time();
	}

	// CÁLCULO // Estrategia: Dividir T y reducir

	// Pasar datos de entrada
	MPI_Bcast (&N, 1, MPI_INT,
			0, MPI_COMM_WORLD);

	MPI_Bcast (&P, 1, MPI_INT,
			0, MPI_COMM_WORLD);

	N2 = N*N;

	if(rank!=0) {
		x = new unsigned[P];
		y = new unsigned[P];
		e = new float[P];
	}

	MPI_Bcast (x, P, MPI_INT,
			0, MPI_COMM_WORLD);


	MPI_Bcast (y, P, MPI_INT,
			0, MPI_COMM_WORLD);

	MPI_Bcast (e, P, MPI_FLOAT,
			0, MPI_COMM_WORLD);

	const int lN = N/size;  //< números de valores de X locales
	const int lN2 = lN*N;

	// Reservar memoria para los procesos locales
	float* lT = new float[P];

	float* lE_mem = new float[lN2];
	float** lE = new float* [lN];
	for(int i=0; i<lN; i++)
		lE[i] = &lE_mem[i*N];

	for(int p=0; p<P; p++)
		lT[p] = 0;

	for(int X=0; X<lN; X++)
		for(int Y=0; Y<N; Y++)
			lE[X][Y] = 0;

	// Cálculo paralelo
for (unsigned X=0; X<lN; X++)
```

```
        for (unsigned Y=0; Y<N; Y++)
                for(unsigned p=0; p<P; p++){

                        float dx = ((int) X+rank*lN) - ((int)x[p]);
dx *= dx;  // dx = (x-xp)²
                        float dy = ((int) Y) - ((int)y[p]);
dy *= dy;  // dy = (y-yp)²
                        float A_pxy_ = exp(sqrt(dx+dy));

                        float EA_pxy_ = e[p] / ( A_pxy_*N2 );

                lT[p] += EA_pxy_;
                lE[X][Y] += EA_pxy_;
                  }



      MPI_Gather (lE_mem, lN2, MPI_FLOAT,
                      rank==0? E[0]:NULL, lN2, MPI_FLOAT,
                        0, MPI_COMM_WORLD);

      MPI_Reduce(lT, T , P, MPI_FLOAT, MPI_SUM, 0, MPI_COMM_WORLD);


      // MOSTRAR RESULTADOS
      if(rank==0) {
            double t1 = get_time();
            cout << "Tiempo de ejecución = " << t1 - t0 << "s" <<
endl;

            // mostrar valores finales solo en modo prueba
            if(argc < 3) {
                  cout << "T(p)" << endl;
                  for (unsigned p=0; p<P; p++)
                        cout << "T(" << p << ") = " << T[p] << endl;

                  cout<< endl;

                  cout << "E(x,y)" << endl;
                  for (unsigned i=0; i<N; i++) {
                        for (unsigned j=0; j<N; j++)
                              cout << fixed << setprecision(2)
<<E[j][i] << '\t';
                        cout << endl;
                  }
            }
      }

      MPI_Finalize();
```

```
    return 0;
}
```