Sobre el cálculo de trayectorias en la extracción de Estructuras Coherentes Lagrangianas

Rocío Carratalá-Sáez[1], José Sierra-Pallares[2], Diego R. Llanos[1] y Arturo Gonzalez-Escribano[1]

Resumen— **La extracción de Estructuras Coherentes Lagrangianas (LCS) es común en diversos campos de dinámica de fluidos, centrados en estudiar el comportamiento de las partículas que integran determinados flujos presentes en la naturaleza, los cuerpos humanos y animales, determinados fluidos artificiales, etc. En concreto, en el proceso de extracción de LCS se llevan a cabo miles o incluso millones de cálculos de mapas de flujo para poder analizar las trayectorias de las partículas que conforman un determinado flujo. Los procedimentos que realizan dichos cálculos forman el cuello de botella de la extracción de LCS.**

En este trabajo presentamos un estudio detallado de cómo se realiza el cálculo de los mapas de flujo, su implementación y una paralelización sencilla de la misma que utiliza OpenMP para ejecutar nuestro código sobre arquitecturas multihilo. Gracias a que las posiciones que se van estimando en el proceso de cálculo los mapas de flujo son independientes para cada una de las partículas de interés, en cada uno de los instantes de tiempo evaluados, tanto en escenarios 2D como en 3D, se observa una eficiencia paralela destacable.

Palabras clave— **Dinámica de fluidos, Mapas de Flujo, Multihilo, OpenMP.**

I. Introducción

En el campo de dinámica de fluidos, tiene interés la extracción de Estructuras Coherentes Lagrangianas (LCS), dado que permiten estudiar el comportamiento de las partículas que integran diversos flujos. Entre los fenónemos regidos por LCS, destacan aquellos asociados a desechos flotantes (como por ejemplo los generados al producirse derrames de petróleo), flotadores superficiales (como es el caso de los patrones de clorofila en el océano) o partículas depositadas en la atmósfera (por ejemplo, nubes de ceniza volcánica o esporas), entre muchos otros derivados tanto del comportamiento humano como animal [1].

El proceso de extracción de LCS se compone de dos fases principales: el cálculo del mapa de flujo (*flow-map*) y el del Exponente de Liapunov de Tiempo Finito (FTLE), siendo el primero el cuello de botella en lo que a tiempo de ejecución se refiere [2]. Por ello, nosotros nos centramos en el análisis, diseño y optimización del proceso de cálculo del mapa de flujo.

Las principales contribuciones del presente trabajo son:

- Descripción detallada del proceso de cálculo del mapa de flujo.
- Análisis de la mejora del rendimiento que puede ofrecer una paralelización naíf basada en directivas OpenMP.
- Evaluación del tiempo de ejecución al calcular el mapa de flujo de dos flujos habituales en la extracción de LCS: *Double-Gyre* [3] y *Arnold–Beltrami–Childres* [4], con el objetivo de comprobar el rendimiento tanto en espacios bidimensionales como tridimensionales, respectivamente.

Este artículo se estructura como sigue: en la Sec. II se presentan los principales conceptos teóricos necesarios para entender el proceso de cálculo de los mapas de flujo; en la Sec. III se hace un repaso de los trabajos existentes en la literatura con un propósito similar al presentado en este trabajo, incidiendo en las diferencias; en la Sec. IV se presentan los detalles de la implementación desarrollada; en la Sec. V se detallan los experimentos realizados y los resultados obtenidos en los mismos; en la Sec. VI se destacan las principales conclusiones; y, finalmente, en la Sec. VII se plantean las principales líneas de trabajo futuro.

II. Cálculo del mapa de flujo

En el proceso de extraer LCS, se consideran los campos de flujos definidos por $\vec{v}(\vec{x}, t)$, los cuales generan trayectorias a lo largo del sistema dinámico definido del siguiente modo:

$$\dot{\vec{x}} = \vec{v}(\vec{x}, t) \tag{1}$$

con $\vec{x} \in \mathcal{U}$, $t \in [t_0\, t_1]$.

Las posiciones $\vec{x} = (x_1, x_2, x_3)$ varían en un dominio acotado $\mathcal{U} \in \mathbb{R}^{\not\vdash}$, con los valores de tiempo contenidos en un determinado intervalo $[t_0, t_1]$. Se define con ello el siguiente mapa de flujo:

$$F_{t_0}^{t} := \vec{x}(t, t_0, \vec{x}_0), \tag{2}$$

con $\vec{x} \in \mathcal{U}$, $t \in [t_0, t_1]$, tomando una posición inicial $\vec{x}_0$ asociada a la posición $\vec{x}(t; t_0, \vec{x}_0)$ en el instante de tiempo t.

De este modo se *mapea* la posición final de una partícula en un fluido a la posición asociada al instante t según su trayectoria. Tras esto, la LCS se puede obtener mediante el FTLE, definido del siguiente modo:

$$FTLE_{t_0}^{t_1}(\vec{x_0}) = \frac{1}{t_1 - t_0} \log \sqrt{\lambda(\vec{x_0})} \tag{3}$$

donde λ es el máximo valor propio del tensor de Cauchy-Green C:

$$C(\vec{x}_0) = \left[\nabla F_{t_0}^{t_1}(\vec{x}_0)\right]^T \nabla F_{t_0}^{t_1}(\vec{x}_0) \tag{4}$$

[1]Depto. Informática de la Universidad de Valladolid, e-mails: {rocio, diego, arturo}@infor.uva.es.

[2]Depto. Ingeniería Energética y Fluidomecánica de la Universidad de Valladolid, e-mail: jsierra@eii.uva.es.

Debe tenerse en cuenta que en el proceso de extracción de la FTLE pueden llegar a calcularse millones de trayectorias [5] y, en todas ellas, el cálculo del mapa de flujo es el cuello de botella.

Por su parte, el cálculo del mapa de flujo se compone de una operación principal: la resolución del Método de Runge-Kutta de cuarto orden (RK4) para estimar con él la posición de una determinada partícula del flujo para un instante de tiempo dado. Con RK4 se resuelve el Problema de Valor Inicial (IVP) que se describe en la Ecuación II, lo cual se traduce en resolver la siguiente Ecuación Diferencial Ordinaria (ODE):

$$\frac{dx}{dt} = f(x,y), \qquad with\ y(0) = y_0. \tag{5}$$

Este método se implementa como un algoritmo iterativo descrito de la siguiente forma:

$$\begin{aligned} k_1 &= h \cdot f(x_n, y_n) \\ k_2 &= h \cdot f(x_n + \tfrac{h}{2}, y_n + \tfrac{k_1}{2}) \\ k_3 &= h \cdot f(x_n + \tfrac{h}{2}, y_n + \tfrac{k_2}{2}) \\ k_4 &= h \cdot f(x_n + h, y_n + k_3) \\ y_{n+1} &= y_n + \tfrac{1}{6}(k_1 + 2k_2 + 2k_3 + k_4) \end{aligned} \tag{6}$$

En particular, cada f utilizada en el método RK4, dado que las velocidades son desconocidas, se sustituye por una interpolación lineal que permite estimar la velocidad en el instante t de tiempo para una determinada partícula, en base a las velocidades v_i y vj que se conocen asociadas a los tiempos t_i y t_j, con $t_i < t < t_j$. Para el caso de dos dimensiones, esto es:

$$\begin{aligned} (vx, vy) = (&\tfrac{vx_i \cdot (t_j - t) + vx_j \cdot (t_j - t_i)}{t_j - t_i}, \\ &\tfrac{vy_i \cdot (t_j - t) + vy_j \cdot (t_j - t_i)}{t_j - t_i}) \end{aligned} \tag{7}$$

El caso tridimensional es equivalente.

Cabe notar, no obstante, que no solamente la velocidad es desconocida. En algunos casos, también lo será el punto sobre el que se va a calcular la posición final, es decir, este no pertenecerá al conjunto de puntos que definen el mallado formado por los datos conocidos. En esos casos, será necesario estimar, además, los valores de v_i y v_j.

La información originalmente conocida sobre el flujo viene dada en forma de mallado. En nuestro caso particular, estos mallados están compuestos por puntos que definen triángulos en el caso bidimensional y tetraedros en el tridimensional. La estimación de los valores de velocidad mencionados (v_i y v_j) sobre un punto desconocido es posible gracias al uso de Coordenadas Baricéntricas [6]. Estas pueden interpretarse como *el peso o la importancia relativa* de cada uno de los vértices del triángulo o tetraedro con respecto a un punto contenido en el mismo; dicho de otro modo, son los valores escalares que permiten expresar un punto contenido en un triángulo o tetraedro como *combinación lineal* de los vértices del mismo.

Formalmente, las Coordenadas Baricéntricas se expresan del modo que sigue. En un espacio bidimensional, sea el triángulo definido por los vértices V_1, V_2 y V_3, y sea el punto $P = (x, y)$ contenido en el mismo, entonces:

$$P = (x,y) = \lambda_1 \cdot V_1 + \lambda_2 \cdot V_2 + \lambda_3 \cdot V_3, \tag{8}$$

con $\lambda_1 + \lambda_2 + \lambda_3 = 1$ y $0 \leq \lambda_1, \lambda_2, \lambda_3 \leq 1$. La expresión en el caso tridimensional es equivalente a esta, considerando una dimensión más.

Así, dadas las Coordenadas Cartesianas de un punto $P = (x, y)$ contenido en un triángulo definido por los vértices $V_1 = (x_1, y_1)$, $V_2 = (x_2, y_2)$ y $V_3 = (x_3, y_3)$, las Coordenadas Baricéntricas que permiten expresar P en función de esos tres vértices son:

$$\begin{aligned} \lambda_1 &= \frac{(y_2 - y_3) \cdot (x - x_3) + (x_3 - x_2) \cdot (y - y_3)}{(y_2 - y_3) \cdot (x_1 - x_3) + (x_3 - x_2) \cdot (y_1 - y_3)} \\ \lambda_2 &= \frac{(y_3 - y_1) \cdot (x - x_3) + (x_1 - x_3) \cdot (y - y_3)}{(y_2 - y_3) \cdot (x_1 - x_3) + (x_3 - x_2) \cdot (y_1 - y_3)} \\ \lambda_3 &= 1 - \lambda_1 - \lambda_2 \end{aligned} \tag{9}$$

El caso tridimensional es análogo.

III. Estado del arte

La extracción de LCS es un proceso que ha suscitado el interés tanto en la comunidad de ingeniería mecánica y física como en las ciencias computacionales. En lo que respecta al primer grupo, destacan los trabajos de Onu *et al.* [5], Brunton y Rowley [2], Conti *et al.* [7], Finn y Apte [8] y Ameli *et al.* [9]. En el campo de la computación, las principales contribuciones se han hecho por parte de Garth *et al.* [10], Sadlo y Peikert [11] y Garth *et al.* [12]. A continuación destacamos las principales diferencias de nuestro trabajo con respecto a los que acabamos de listar.

Conti *et al.* [7] presenan una implementación que acelera el cálculo del FTLE usando OpenCL y aprovechando tanto CPU como GPU. Aunque nosotros nos limitamos a paralelismo multinúcleo y no ofrecemos soporte para GPU por el momento, el hecho de usar OpenMP en lugar de OpenCL supone una ventaja en términos de eficiencia porque evitamos el *overhead* derivado de OpenCL, causado por su propósito más generalizado para dar soporte a distintos dispositivos. Además, nuestro trabajo pone en foco en el cálculo del mapa de flujo, en lugar del FTLE.

Finn y Apte [8] presentan diferentes aceleraciones en lo que al cálculo del FTLE respecta, reduciendo el posprocesado de los campos de velocidad y acelerando el cálculo de LCS. De nuevo, nuestra principal diferencia es que nos centramos en analizar en profundidad el cálculo de los mapas de flujo y paralelizarlos.

Ameli *et al.* [9] proporcionan la única implementación disponible en código abierto que hemos encontrado para la extracción de LCS: GitHub[1]. Nuestra

[1] `https://github.com/FlowPhysics/FlowTK`

principal ventaja frente a este trabajo es que dicha implementación es secuencial.

Garth *et al.* [10], si bien ofrecen una aceleración del cómputo basada en el aprovechamiento de GPU, únicamente abordan espacios bidimensionales, mientras que nuestro trabajo también cubre espacios tridimensionales.

Sadlo y Peikert [11] presentan un algoritmo eficiente para el cálculo del FTLE en espacios tridimensionales, pero hay una escasa descripción del mismo y el código no es público, al contrario que el nuestro.

Garth *et al.* [12] muestran algunas mejoras algorítmicas basadas en la reducción del número de trayectorias necesarias para la obtención del FTLE, si bien no llegan a publicar una implementación de las mismas, ni tampoco una propuesta paralela.

Por tanto, nuestro trabajo aúna como factores diferenciadores:

1. se centra en el cálculo de los mapas de flujo, que constituyen el cuello de botella en la extracción de LCS;
2. ofrecemos una descripción detallada de dicho cálculo;
3. cubrimos tanto espacios bidimensionales como tridimensionales;
4. aprovechamos OpenMP para paralelizar nuestra implementación;
5. nuestra implementación es de código abierto.

IV. Implementación del cálculo del mapa de flujo

La implementación que presentamos en este trabajo lleva a cabo el cálculo de los mapas de flujo tanto en espacios bidimensionales como tridimensionales. Hemos diseñado un código en lenguaje C que realiza este cálculo valiéndose además de OpenMP para realizar ejecuciones paralelas sobre arquitecturas multihilo.

A. Datos de entrada

Los datos originalmente conocidos del flujo a analizar se proporcionan mediante ficheros de tipo VTK, los cuales se han generado mediante el software de código abierto OpenFOAM [13], que es el de referencia en el campo de Dinámica de Fluidos Computacional. Estos ficheros contienen una descripción del mallado que abstrae y recoge la información conocida *a priori* sobre el flujo a analizar.

Con el fin de procesar estos ficheros, hemos diseñado unos códigos sencillos en Python que extraen la información útil para el cálculo de los mapas de flujo. Estos *scripts* generan cuatro ficheros: `coords.txt`, `faces.txt`, `times.txt` y `velocity.txt` que constituyen parte de los datos de entrada proporcionados a nuestra implementación en C. Una vez leídos al comienzo de nuestro código para calcular los mapas de flujo, se rellena una estructura de tipo `mesh_t` que recoge toda la información relacionada con el mallado (dimensión del espacio, número de puntos, número de caras/volúmenes, número de instantes tiempos para los cuales se conocen valores de velocidad, puntos, caras/volúmenes e instantes de tiempo) y varias de tipo `point_t` recogiendo, para cada punto del mallado, todos sus datos (coordenadas y valores de velocidad conocidos). Estas estructuras se describen en la Fig. 1.

```
typedef struct Point {
   int      index; /* índice en mesh->points */
   double  *coordinates;
   double  *velocity;
} point_t;

typedef struct Mesh {
   int      nDim;
   int      nPoints;
   int      nFaces;
   int      nTimes;
   point_t *points;
   int     *faces;
   double  *times;
} mesh_t;
```

Fig. 1: Estructuras de datos diseñadas para almacenar la información originalmente conocida, en forma de mallado (`mesh_t`) compuesto por puntos o vértices (`point_t`).

B. Funciones principales

El cálculo de los mapas de flujo, tal como se ha detallado en la Sec. II, se realiza mediante resoluciones del método RK4. En concreto, se resuelve una llamada a dicho método por cada punto del mallado y cada instante de tiempo para el que se quiere estimar la trayectoria del punto. En la Fig. 2 se muestra el esquema de la función principal, encargada de llevar a cabo dicho procedimiento. Como datos de entrada se proporcionan los ficheros detallados en la sección anterior, así como la dimensión del espacio, el primer instante de tiempo a evaluar y el último (`t0` y `tf`), la diferencia entre cada instante de tiempo a evaluar (`tdelta`) y el número de iteraciones a realizar en el cálculo de RK4. El primer paso de este procedimiento es leer los datos del mallado a partir de los ficheros de entrada y, a continuación, comienza el cálculo de las trayectorias en sí.

La implementación del método RK4 es una traducción directa de lo que se ha presentado en la Sec. II, por lo que no hemos considerado necesario detallarla.

En cambio, sí que consideramos importante explicar el procedimiento que se sigue para interpolar la velocidad en aquellos puntos cuyos valores de velocidad para ti, tj son desconocidos porque no pertenecen al mallado. La Fig. 3 muestra un esquema de este procedimiento en el caso bidimensional, al que se le llama siempre al calcular $k2$, $k3$ y $k4$ en las llamadas a RK4, donde se interpola una velocidad sobre un punto ligeramente desplazado con respecto a las coordenadas del punto pasadas como parámetro de entrada y utilizadas directamente en el cálculo de $k1$. Tal como se ha explicado en la Sec. II, este procedimiento encuentra primero la cara a la que pertenece el punto desconocido, tras esto calcula las Coordenadas Baricéntricas que permiten expresarlo en función de los vértices del triángulo que forma dicha cara, a continuación se interpola la velocidad de dichos vértices para el instante t y, finalmente, se estima la velocidad del punto en el instante t utili-

```
1  double * Compute_Flowmap(FILE coords,FILE faces,
2                           FILE times, FILE vels,
3                           int nDim,
4                           double t0, double tf,
5                           double tdelta,
6                           int stepsRK4)
7  {
8      int i, j, idx;
9      double t, ti, tj;
10     point_t P;
11     double vi[2];
12
13     /* Leer información original del mallado */
14     mesh_t Mesh = New_Mesh(coords, faces, times,
15                            vels, nDim);
16
17     /* Calcular el mapa de flujo */
18     double res[nDim*mesh->nPoints*mesh->nTimes];
19     for (i=0; i<mesh->nPoints; i++)
20     {
21         P = mesh->points[i];
22         j = 0;
23         while (t0+j*tdelta < tf)
24         {
25             t = t0 + j * tdelta;
26             ti = Find_Previous_t(mesh, t);
27             vi = 2D_Get_Vel(mesh, P, ti);
28             idx = i*mesh->nTimes*mesh->nDim +
29                   j*mesh->nDim;
30
31             /* Estimar vel para P en t (RK4) */
32             result[idx] = RK4(mesh, P, vi,
33                               t, t+tdelta,
34                               stepsRK4);
35         }
36     }
37     return res;
38 }
```

Fig. 2: Esquema del procedimiento principal encargado de calcular el mapa de flujo utilizando RK4 para aproximar las diferentes trayectorias.

zando las Coordenadas Baricéntricas calculadas y las distintas velocidades interpoladas para cada vértice.

C. Paralelización con OpenMP

Este algoritmo es *embarrassingly parallel*, dado que cada llamada a RK4 en el procedimiento ilustrado en la Fig. 2 es independiente del resto. Por ello, en este trabajo analizamos el rendimiento de una paralelización naíf usando directivas OpenMP que simplemente consiste en paralelizar el bucle externo de dicho algoritmo; es decir, colocar la directiva `#pragma omp parallel for` justo antes del bucle for situado en la línea 18 de la Fig. 2.

En la siguiente sección presentamos una evaluación de rendimiento de dicha paralelización probando distintas políticas de `scheduling` (concretamente, `static`, `dynamic` y `guided`).

V. Resultados experimentales

Los experimentos presentados en esta sección se han llevado a cabo en el servidor `gorgon`, propiedad de la Universidad de Valladolid. Este se compone de dos CPU AMD EPYC 7713 (Ryzen 3) @ 2.0GHz, con 64 Core Processors y 128 hilos físicos cada uno, con un total de 256 unidades físicas de cómputo.

Los códigos se han compilado con `gcc 11.1` y `clang` de `AOCC 3.1.0` (AMD Optimizing C/C++ and Fortran Compilers), usando en ambos casos el flag `-march=znver3` para generar instrucciones que se ejecuten en la arquitectura EPYC/RYZEN de tercera generación de manera óptima y `-O3`.

```
1  double* 2D_Interp_Vel_Unknown_P(mesh_t mesh,
       double *P, double t)
2  {
3      int d;
4      double vel[2];
5      double veli[2], velj[2];
6      double vel_v1[2], vel_v2[2], vel_v3[2];
7      double bcoords[3];
8      double ti, tj;
9
10     /* Buscar cara que contiene el punto P */
11     int face = Find_Face(mesh, P);
12
13     /* Si se encuentra la cara, interpolar */
14     if ( face >= 0 )
15     {
16         /* Obtener Coords. Baricéntricas de P */
17         bcoords = 2D_Bary_Coords(mesh, face, P);
18
19         /* Fijar ti < t < tj */
20         ti = Find_Previous_t (mesh, t);
21         tj = Find_Next_t (mesh, t);
22
23         /* Interp. vel para P en ti */
24         vel_v1 = 2D_Get_Vel(mesh, face[0], ti);
25         vel_v2 = 2D_Get_Vel(mesh, face[1], ti);
26         vel_v3 = 2D_Get_Vel(mesh, face[2], ti);
27         veli[0] = bcoords[0] * vel_v1[0] +
28                   bcoords[1] * vel_v2[0] +
29                   bcoords[2] * vel_v3[0];
30         veli[1] = bcoords[0] * vel_v1[1] +
31                   bcoords[1] * vel_v2[1] +
32                   bcoords[2] * vel_v3[1];
33
34         /* Interp. vel para P en tj */
35         vel_v1 = 2D_Get_Vel(mesh, face[0], tj);
36         vel_v2 = 2D_Get_Vel(mesh, face[1], tj);
37         vel_v3 = 2D_Get_Vel(mesh, face[2], tj);
38         velj[0] = bcoords[0] * vel_v1[0] +
39                   bcoords[1] * vel_v2[0] +
40                   bcoords[2] * vel_v3[0];
41         velj[1] = bcoords[0] * vel_v1[1] +
42                   bcoords[1] * vel_v2[1] +
43                   bcoords[2] * vel_v3[1];
44
45         /* Calcular vel para P en t */
46         vel = 2D_Linear_Interp (ti, tj,
47                                 veli, velj);
48     }
49     return vel;
50 }
```

Fig. 3: Esquema del procedimiento para interpolar la velocidad (`vel`) en un punto (`P`) que no pertenece al mallado, en un instante de tiempo determinado (`t`).

Como caso de prueba se ha tomado el flujo bidimensional Double-Gyre [3] (habitual en fluidos en geofísica) y el tridimensional Arnold–Beltrami–Childress (ABC) [4], resultante de resolver la ecuación de Euler.

A. Evaluación en 2D con el flujo Double-Gyre

Para el flujo bidimensional Double-Gyre, se ha evaluado el tiempo de ejecución total cuando se calculan velocidades para 500 y 750 instantes de tiempo con 50, 100, 150, 200 y 250 hilos (además del caso secuencial), usando los compiladores GCC y AOCC y, con cada uno de ellos, tres políticas diferentes para el *scheduler* de OpenMP: `static`, `dynamic` y `guided`. En la Fig. 4 se muestran los tiempos de ejecución para las pruebas realizadas en el caso 2D y la Fig. 5 muestra el correspondiente speedup.

A partir de los resultados obtenidos se observa:

- El speedup máximo alcanzado al calcular la velocidad en 500 instantes de tiempo usando 250 hilos para la versión compilada con GCC es de `76.5x`, siendo de `82.5x` para la versión compi-

#t	Comp Sched	Secuencial	50th	100th	150th	200th	250th
500	GCC Static	220.29214	6.44629	5.51211	4.59786	3.73640	3.60236
500	GCC Dynamic	220.29214	6.23031	5.17530	3.77477	3.22336	2.87808
500	GCC Guided	220.29214	6.78486	7.76955	5.68125	4.14000	3.83605
500	AOCC Static	251.09141	6.60805	6.08429	4.80994	3.76779	3.35219
500	AOCC Dynamic	251.09141	6.41420	4.38371	3.78535	3.39456	3.35725
500	AOCC Guided	251.09141	6.34409	5.44692	4.24300	3.40308	3.04372

#t	Comp Sched	Secuencial	50th	100th	150th	200th	250th
750	GCC Static	329.48524	9.54349	7.96529	6.76850	5.32805	4.54874
750	GCC Dynamic	329.48524	9.03900	7.33890	5.54946	4.24207	4.19063
750	GCC Guided	329.48524	10.28556	11.49227	8.47389	6.38040	5.88543
750	AOCC Static	371.79073	9.79150	7.37773	6.62882	5.97222	4.86277
750	AOCC Dynamic	371.79073	9.53176	7.10015	5.57287	5.08337	5.04232
750	AOCC Guided	371.79073	9.59944	6.00914	5.66873	5.23861	4.47321

Fig. 4: Tiempos de ejecución para el caso bidimensional (Double-Gyre). La primera columna representa la cantidad de tiempos para los que se ha calculado la velocidad (500 y 750), la segunda el compilador y la política de scheduling utilizados, las siguientes los tiempos de ejecución respectivamente para el caso secuencial, 50, 100, 150, 200 y 250 hilos en `gorgon`.

#t	Comp Sched	Secuencial	50th	100th	150th	200th	250th
1000	GCC Static	569.5519	15.3976	10.4634	7.8228	6.0341	5.7293
1000	GCC Dynamic	569.5519	13.9141	9.5115	6.8768	5.3174	4.8622
1000	GCC Guided	569.5519	14.7232	10.9947	8.0586	5.9088	5.2663
1000	AOCC Static	759.9312	22.3363	15.0208	10.9592	10.0816	8.6489
1000	AOCC Dynamic	759.9312	19.7764	12.4603	10.4680	9.5634	8.6100
1000	AOCC Guided	759.9312	18.5825	12.5590	10.0904	8.7896	8.3302

#t	Comp Sched	Secuencial	50th	100th	150th	200th	250th
1500	GCC Static	854.4954	23.0440	15.4464	11.7336	9.0121	8.1985
1500	GCC Dynamic	854.4954	20.7729	13.9387	10.0225	7.8567	6.9437
1500	GCC Guided	854.4954	22.0412	15.0770	10.8001	8.8699	8.4815
1500	AOCC Static	1106.5807	32.0124	21.6986	18.0845	14.5889	12.5430
1500	AOCC Dynamic	1106.5807	28.6201	19.6684	16.0479	14.2467	12.5492
1500	AOCC Guided	1106.5807	28.2924	16.9562	14.1231	12.9822	12.2908

Fig. 6: Tiempos de ejecución para el caso tridimensional (ABC). La primera columna representa la cantidad de tiempos para los que se ha calculado la velocidad (1000 y 1500), la segunda el compilador y la política de scheduling utilizados, las siguientes los tiempos de ejecución respectivamente para el caso secuencial, 50, 100, 150, 200 y 250 hilos en `gorgon`.

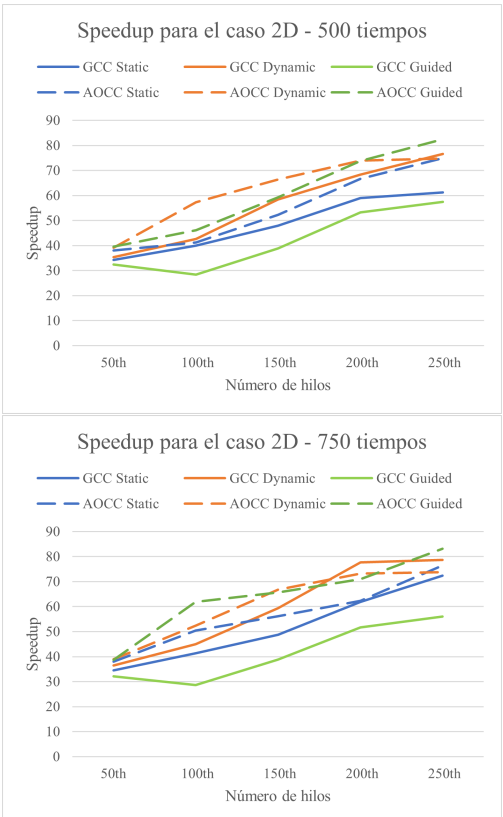

Fig. 5: Speedup para el caso bidimensional (Double-Gyre) y velocidades calculadas para 500 (arriba) o 750 instantes de tiempo (abajo), según los tiempos de ejecución, políticas de scheduling, compiladores y número de hilos mostrados en la Fig. 4.

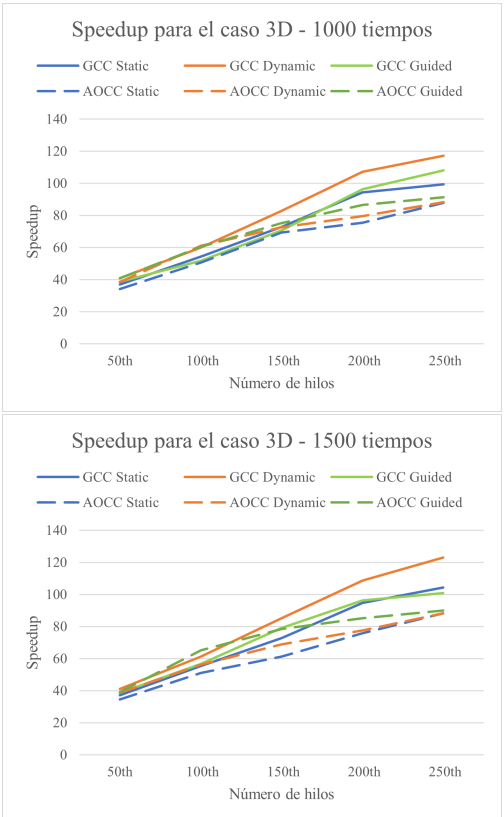

Fig. 7: Speedup para el caso tridimensional (ABC) y velocidades calculadas para 1.000 (arriba) o 1.500 instantes de tiempo (abajo), según los tiempos de ejecución, políticas de scheduling, compiladores y número de hilos mostrados en la Fig. 6.

lada con AOCC. Al aumentar la carga calculando la velocidad en 750 instantes de tiempo este sube ligeramente a `78.6x` para GCC y `83.1x` para AOCC.

- En lo que a compiladores se refiere, con la paralelización naíf planteada, el aprovechamiento máximo de la máquina es de un 30.7 % aproximadamente con GCC y un poco superior en el caso de AOCC, situándose alrededor del 32.5 %.
- Con independencia de la política de *scheduling* de OpenMP elegida, AOCC tiende a ofrecer mejores resultados que GCC.
- Particularmente en lo que a políticas de *scheduling* de OpenMP se refiere, con GCC los mejores resultados siempre los ofrece `dynamic`, mientras que con AOCC hay una escasa diferencia en general entre esta y `guided`, inclinada a favor de esta última al utilizar los 250 hilos.

B. *Evaluación en 3D con el fujo ABC*

Para el flujo tridimensional ABC, se ha evaluado el tiempo de ejecución total cuando se calculan velocidades para 1000 y 1500 instantes de tiempo. Se han probado el mismo número de hilos, políticas de *scheduling* de OpenMP y compiladores que en el caso bidimensional. En la Fig. 6 se muestran los tiempos de ejecución para las pruebas realizadas en el caso 3D y la Fig. 7 muestra el correspondiente speedup.

A partir de los resultados obtenidos se observa:

- El speedup máximo alcanzado al calcular la ve-

locidad en 1000 instantes de tiempo usando 250 hilos para la versión compilada con GCC es de `108.2x`, siendo de `91.2x` para la versión compilada con AOCC. Al aumentar la carga calculando la velocidad en 1500 instantes de tiempo este sube ligeramente a `123x` para GCC y `90x` para AOCC.

- En lo que a compiladores se refiere, con la paralelización naíf planteada, el aprovechamiento máximo de la máquina es de un 48 % aproximadamente con GCC y un poco inferior en el caso de AOCC, situándose alrededor del 35.6 %.
- Con independencia de la política de *scheduling* de OpenMP elegida, AOCC tiende a ofrecer peores resultados que GCC.
- Particularmente en lo que a políticas de *scheduling* de OpenMP se refiere, con GCC los mejores resultados siempre los ofrece `dynamic`, mientras que con AOCC es mejor optar por `guided`.

VI. Conclusiones

En este trabajo hemos presentado detalladamente cómo se lleva a cabo el cálculo de los mapas de flujo, dentro del contexto de extracción de LCS, para el cual suponen el cuello de botella.

Hemos evaluado el rendimiento de una paralelización naíf usando OpenMP y hemos llegado a la conclusión de que, en un servidor como el utilizado, equipado con una arquitectura EPYC/RYZEN de tercera generación, hay diferencias destacables entre lo que puede concluirse para ejecuciones en espacios bidimensionales frente a lo equivalente en tridimensionales.

Por un lado, en 2D, el compilador AOCC combinado con las políticas de *scheduling* de OpenMP `dynamic` o `guided` ofrece los mejores resultados, llegando a aprovecharse hasta un 32.5 % de la máquina usándola en su totalidad.

Por otro lado, en 3D, el compilador GCC combinado con la política de *scheduling* de OpenMP `dynamic` presenta los mejores resultados, llegando a aprovecharse hasta un 48 % de la máquina usándola en su totalidad.

VII. Trabajo Futuro

Como parte del trabajo futuro planeamos: ser capaces de procesar mallados híbridos que contengan caras/volúmenes diferentes; diseñar e implementar mejoras algorítmicas; mejorar la eficiencia paralela en arquitecturas multinúcleo variando la granularidad de las tareas que realiza cada hilo; incoporar soporte para GPU; y explorar el paralelismo sobre memoria distribuida para aprovechar sistemas multinúcleo mediante MPI.

Agradecimientos

Este trabajo se ha financiado con los proyectos PCAS (TIN2017-88614-R) y Project PROPHET-2 (VA226P20) de la Consejería de Educación de la Junta de Castilla y León, Ministerio de Economía, Industria y Competitividad de España, y el programa European Regional Development Fund (ERDF). Jose Sierra-Pallares ha sido financiado por el proyecto VA182P20 de la Junta de Castilla y León.

Referencias

[1] G Haller, "Lagrangian coherent structures," *Annual Review of Fluid Mechanics*, vol. 47, pp. 137–162, 2015.

[2] Steven Brunton and Clarence Rowley, "Fast computation of finite-time lyapunov exponent fields for unsteady flows," *Chaos*, vol. 20, no. 1, pp. 017503, 3 2010.

[3] C. Coulliette and S. Wiggins, "Intergyre transport in a wind-driven, quasigeostrophic double gyre: An application of lobe dynamics," *Nonlinear Processes in Geophysics*, vol. 7, no. 1/2, pp. 59–85, 2000.

[4] Xiao-Hua Zhao, Keng-Huat Kwek, Ji-Bin Li, and Ke-Lei Huang, "Chaotic and resonant streamlines in the abc flow," *SIAM Journal on Applied Mathematics*, vol. 53, no. 1, pp. 71–77, 1993.

[5] K Onu, F Huhn, and G Haller, "Lcs tool: A computational platform for lagrangian coherent structures," *Journal of Computational Science*, vol. 7, pp. 26–36, 3 2015.

[6] Harold Scott Macdonald Coxeter, *Introduction to geometry*, New York, London, 1961.

[7] Christian Conti, Diego Rossinelli, and Petros Koumoutsakos, "Gpu and apu computations of finite time lyapunov exponent fields," *Journal of Computational Physics*, vol. 231, no. 5, pp. 2229–2244, 2012.

[8] Justin Finn and Sourabh V Apte, "Integrated computation of finite-time lyapunov exponent fields during direct numerical simulation of unsteady flows," *Chaos: An Interdisciplinary Journal of Nonlinear Science*, vol. 23, no. 1, pp. 013145, 2013.

[9] Siavash Ameli, Yogin Desai, and Shawn C Shadden, "Development of an efficient and flexible pipeline for lagrangian coherent structure computation," in *Topological Methods in Data Analysis and Visualization III*, pp. 201–215. Springer, 2014.

[10] Christoph Garth, Guo-Shi Li, Xavier Tricoche, Charles D Hansen, and Hans Hagen, "Visualization of coherent structures in transient 2d flows," in *Topology-Based Methods in Visualization II*, pp. 1–13. Springer, 2009.

[11] Filip Sadlo and Ronald Peikert, "Efficient visualization of lagrangian coherent structures by filtered amr ridge extraction," *IEEE Transactions on Visualization and Computer Graphics*, vol. 13, no. 6, pp. 1456–1463, 2007.

[12] Christoph Garth, Florian Gerhardt, Xavier Tricoche, and Hagen Hans, "Efficient computation and visualization of coherent structures in fluid flow applications," *IEEE Transactions on Visualization and Computer Graphics*, vol. 13, no. 6, pp. 1464–1471, 2007.

[13] H.G. Weller, Gavin Tabor, Hrvoje Jasak, and Christer Fureby, "A tensorial approach to computational continuum mechanics using object orientated techniques," *Computers in Physics*, vol. 12, pp. 620–631, 11 1998.