UNIVERSITAT POLITÈCNICA DE VALÈNCIA

Escuela Técnica Superior de Ingeniería del Diseño

IUMPA Instituto Universitario de Matemática Pura y Aplicada

UNIVERSITAT POLITÈCNICA DE VALÈNCIA

Escuela Técnica Superior de Ingeniería del Diseño

Máster Universitario en Ingeniería Aeronáutica

Trabajo Fin de Máster

Desarrollo de herramientas computacionales para la identificación de vórtices en un canal turbulento

Autor:
José Carlos Jaén Ruiz

Tutor:
D. Sergio Hoyas Calvo

Tutor Externo:
Jose Javier Aguilar Fuertes

Cotutor:
D. Lluis García Raffi

Valencia, Septiembre 2020

Mi agradecimiento especial va para mi familia

Agradecimientos

Estos agradecimientos van directamente dedicados a mi familia, especialmente a mis padres, Ana y José, y a mi hermana, Cristina, los cuales han supuesto un gran apoyo durante el proceso de aprendizaje que han supuesto estos seis años, estando conmigo tanto en momentos felices como otros más difíciles, pero gracias a ellos he podido a llegar a ser quien soy. No me puedo olvidar de Ana, la cual ha supuesto un pilar fundamental para la realización de este trabajo, creyendo siempre que puedo conseguir todo lo que me proponga.

A mis amigos por sus incansables muestras de apoyo durante todo este camino, con mención especial a mi compañero de despacho, Kiko, con el que he pasado la mayor parte del tiempo de esta cuarentena, haciendo más amena la realización de este proyecto.

Finalmente, agradezco a mis tutores, especialmente a Sergio, el cual, a pesar de las dificultades debidas a esta pandemia, siempre ha estado dispuesto a ayudar en todo lo que necesitábamos. Gracias a él por su paciencia y comprensión, y por ayudarme a comprender un poco mejor el mundo que nos rodea.

Resumen

El principal objetivo de este Trabajo Final de Máster es el desarrollo de herramientas que nos permitan identificar vórtices en simulaciones numéricas directas, DNS. Para ello se ha hecho uso de distintos algoritmos implementados en Fortran, paralelizados mediante MPI y OpenMP.

A partir de los resultados obtenidos por el código LISO (encargado de resolver la DNS), se identifican los puntos que superan un cierto criterio de vorticidad, generando así una matriz booleana. En primer lugar, se ha desarrollado un algoritmo de identificación de estructuras, el cual divide el dominio entre todos los procesadores, y se hace la identificación a partir de encontrar un punto de origen del vórtice, y recorriéndolo en las 6 direcciones cartesianas. Cuando todos los procesadores han hecho su identificación, se unifican los resultados identificando los vórtices totales del campo.

Una vez tenemos los vórtices identificados en cada instante temporal, se hace el seguimiento temporal de los mismos. Para ello se comparan las estructuras en dos instantes temporales consecutivos, determinando la evolución, vida, muerte, rotura o coalescencia de los vórtices. Finalmente, estas herramientas han sido implementadas como un módulo del código LISO, realizando la identificación y seguimiento de los vórtices al mismo tiempo que se realiza la simulación numérica directa del canal turbulento.

Palabras clave: DNS, Vórtices, Turbulencia, Paralelización.

Resum

El principal objectiu d'aquest Treball Final de Màster és el desenvolupament de ferramentes que ens permeten identificar vòrtexs en simulacions numèriques directes, DNS. Per a això s'ha fet ús de distints algoritmes implementats en Fortran, parallelitzats per mitjà de MPI i OpenMP.

A partir dels resultats obtinguts pel codi LISO (encarregat de resoldre la DNS), s'identifiquen els punts que superen un cert criteri de vorticitat, generant així una matriu booleana. En primer lloc, s'ha desenvolupat un algoritme d'identificació d'estructures, el qual dividix el domini entre tots els processadors, i es fa la identificació trobant un punt d'origen del vòrtex, i recorrent-ho en les 6 direccions cartesianes. Quan tots els processadors han fet la seua identificació, s'unifiquen els resultats identificant els vòrtexs totals del camp.

Una vegada tenim els vòrtexs identificats en cada instant temporal, es fa el seguiment temporal dels mateixos. Per a això es comparen les estructures en dos instants temporals consecutius, determinant l'evolució, vida, mort, ruptura o coalescencia dels vòrtexs. Finalment, estes ferramentes han sigut implementades com un mòdul del codi LISO, realitzant la identificació i seguiment dels vòrtexs alhora que es realitza la simulació numèrica directa del canal turbulent.

Paraules Clau: DNS, Vòrtices, Turbulència, Parallelització.

Abstract

The main objective of this Master Thesis is the development of tools that allow the identification of vortices in direct numerical simulations (DNS). For that purpose, different algorithms implemented in Fortran has been used and parallelized using MPI and OpenMP.

From the results obtained from LISO code (focused on solving DNS), points that accomplish a certain vorticity criterion are identified with which a boolean matrix is generated. First, a vortex identification algorithm has been developed, which divides the domain between all the processors. Each processor identifies a germ point from which the rest of the vortex is found, traveling through it in the 6 Cartesian directions. When all the processors have made their identification, the results are unified by identifying the total vortices of the field.

Once the vortices have been identified at each time step, they are tracked over time. To this end, the structures are compared in two consecutive time instants, determining the evolution, life, death, rupture or coalescence of the vortices. Finally, these tools have been implemented as a module of the LISO code, carrying out the identification and tracking of the vortices at the same time that the direct numerical simulation of the turbulent channel is carried out.

Keywords: DNS, Vortex, Turbulence, Parallelization.

Índice general

Índice de figuras

Índice de cuadros

Capítulo 1

Introducción

El objetivo principal de este Trabajo Final de Máster es el desarrollo y optimización de algoritmos para el estudio de estructuras coherentes o vórtices en un canal resuelto mediante una simulación numérica directa o DNS. La turbulencia es un régimen en el que el campo de velocidades del flujo varía de forma importante y aleatoria tanto en espacio como en tiempo. Aunque esta definición pueda parecer complicada, todos los flujos turbulentos tienen que cumplir las siguientes características.

- **Irregularidad:** tal y como indica su propia definición, los flujos turbulentos son aleatorios e irregulares. La connotación más importante de esta característica es que su estudio analítico se hace muy complicado, y normalmente se recurren a métodos estadísticos para su resolución.

- **Tridimensional:** los flujos turbulentos son puramente 3D. Una de las características más importantes es que tienen altos niveles de vorticidad fluctuante. El principal mecanismo que mantiene la vorticidad, conocido por *Vortex Stretching*, no existe en flujos bidimensionales.

- **Difusividad:** esta es la característica más importante de la turbulencia. La turbulencia hace que aumenten las tasas de transferencia de calor, masa y energía. Esta característica es la responsable de fenómenos como la prevención de separación de la capa límite a altos ángulos de ataque o la resistencia al movimiento del flujo en tuberías. Esta característica no es siempre negativa, ya que hay aplicaciones en las que se busca incrementar la turbulencia para favorecer la mezcla, como ocurre en la inyección de combustible diesel en un motor de combustión interna.

- **Continuidad:** los flujos turbulentos son continuos, es decir, las escalas más pequeñas de la turbulencia son órdenes de magnitud superiores a las moleculares. Aunque ya se verá mas adelante, la escala más pequeña del flujo turbulento es la escala de Kolmogorov (η), y la escala molecular más relevante es la del camino libre medio (ξ). Ambas escalas están relacionadas por el número de Knudsen.

$$\frac{\xi}{\eta} \sim \frac{M}{Re^{1/4}} \tag{1.1}$$

Donde M es el número de Mach y Re es el número de Reynolds. Este parámetro suele ser muy pequeño excepto en casos extremos, como podría ser una nebulosa gaseosa.

De forma general, podemos diferenciar a un flujo turbulento a partir de su número de Reynolds. Este número fue descrito por Osborne Reynolds [1] en 1894. Tiene la siguiente formulación,

$$Re = \frac{u\,L}{\nu}, \tag{1.2}$$

donde u es la velocidad, L es una longitud característica y ν es la viscosidad cinemática. Este número se puede ver como una relación entre la fuerzas inerciales, en el numerador de la ecuación, y fuerzas viscosas, en el denominador. Cuando las fuerzas viscosas dominan a las fuerzas inerciales (bajos números de Reynolds) dan lugar a lo que se conocen como fluidos laminares, en los que el flujo se mueve en capas que no se mezclan. Estos flujos suelen tener solución analítica en geometrías sencillas, por lo que son más fáciles de estudiar. Sin embargo, cuando las fuerzas inerciales dominan a las fuerzas viscosas (altos números de Reynolds), se producen inestabilidades que llevan a la rotura de la ordenación en capas, formando torbellinos con un fuerte carácter caótico.

Estas transiciones entre flujo laminar y turbulento se producen aproximadamente a $Re \approx 2000$ en tuberías y $Re \approx 5 \cdot 10^5$ para flujos sobre placas planas. Para hacernos una idea, el número de Reynolds en un planeador es del orden de $Re \sim 1.6 \cdot 10^5$, o en el caso del Boeing 747, $Re \sim 2 \cdot 10^9$. Por norma general, en ingeniería los flujos se producen a elevados números de Reynolds, por lo que los flujos turbulentos no son la excepción, sino la regla.

La turbulencia, y sobre todo la turbulencia de pared, tiene una importancia tecnológica muy importante. Tal y como estima Jiménez en [2], aproximadamente el 25 % de la energía que se usa en la industria y en el comercio se gasta en mover fluidos a través de tuberías, o mover vehículos sobre el agua o aire, y un cuarto de esa energía es disipada en zonas cercanas a la pared. Por este motivo, un mejor entendimiento de la turbulencia nos puede llevar a un mejor aprovechamiento de la energía, reduciendo su consumo y así las emisiones de CO_2 en el caso de que la energía se obtenga de combustibles fósiles.

Las ecuaciones fundamentales para el estudio de la turbulencia y para el estudio de los fluidos en general son las llamadas ecuaciones de Navier-Stokes, las cuales son ecuaciones en derivadas parciales no lineales que representan el movimiento de una partícula fluida junto a la ecuación de conservación de la masa. Las formulaciones en notación de Einstein es la siguiente:

$$\frac{\partial U_i}{\partial x_i} = 0 \tag{1.3}$$

$$\frac{\partial U_i}{\partial t} + U_j \frac{\partial U_i}{\partial x_j} = -\frac{1}{\rho}\frac{\partial P}{\partial x_i} + \nu \frac{\partial^2 U_i}{\partial x_j^2} + f_i \tag{1.4}$$

La ecuación 1.3 es la ecuación de conservación de la masa para flujo incompresible y 1.4 son las ecuaciones de Navier-Stokes. A este sistema se le puede acoplar de forma natural una ecuación de transporte de la energía o de partículas, pero al no ser tratada en este proyecto, no se han incluido en el sistema.

1.1. Las escalas del flujo turbulento

Como se ha explicado, los flujos turbulentos se producen a altos números de Reynolds, en los cuales los efectos inerciales son muchos más importantes que los viscosos. Esto nos podría llevar a pensar que estos son despreciables en los flujos turbulentos. Sin embargo, las ecuaciones de Navier-Stokes, y en particular los términos no lineales contrarrestan este efecto, generando escalas lo suficientemente pequeñas para ser afectadas por la viscosidad.

Por lo tanto, tenemos en el flujo unas escalas grandes que se encargar de transportar grandes cantidades de energía y de momento, en las que la viscosidad tiene un efecto despreciable; y por otro lado, unas escalas mucho más pequeñas, con unos tiempos característicos también más cortos que los del flujo principal. Con este punto de partida es donde aparecen los conceptos de cascada de energía y las hipótesis de Kolmogorov. Andrey Kolmogorov (1903-1987) fue un matemático y físico ruso que tuvo una gran influencia en el estudio de la turbulencia, ya que sus estudios nos permiten hacer predicciones simples pero fundamentales en esencia. La idea principal es que existe una escala muy pequeña, la cual es universal e independiente del Reynolds y solo depende de la viscosidad y disipación.

Para este análisis vamos a definir las siguientes escalas características de los remolinos:

- **Tamaño:** ℓ
- **Velocidad:** $u(\ell)$
- **Tiempo:** $\tau(\ell) \equiv \ell / u(\ell)$

Según el concepto de la cascada de energía, la energía entra al flujo por las escalas más grandes, es decir, los torbellinos tienen una escala que es comparable a la del flujo.

$$\ell_0 \sim \mathcal{L}, \qquad u_0 \equiv u(\ell_0) \tag{1.5}$$

Estos remolinos grandes son inestables y se rompen en otros más pequeños, transfiriendo la energía a estos remolinos. Los remolinos resultantes también son inestables, y se rompen en otros todavía mas pequeños. Este proceso continúa hasta que se alcanzan unos remolinos con un Reynolds local que es lo suficientemente pequeño como para que los efectos viscosos sean importantes, y la viscosidad lamine el flujo, disipando la energía.

$$Re(\ell) \equiv \frac{u(\ell)\ell}{\nu} \tag{1.6}$$

Este proceso de transferencia de energía de las escalas grandes a las más pequeñas se conoce como *cascada de energía*. Aunque la disipación se hace al final de la cascada, esta queda definida al inicio de la cascada, ya que la energía entra en las grandes escalas, y toda esta energía es la que finalmente se tiene que disipar. Para estimar la disipación, tenemos que la energía de estos remolinos es $\sim u_0^2$. Así, la transferencia de energía se puede determinar como.

$$\frac{u_0^2}{\tau_0} = \frac{u_0^3}{\ell_0} \longrightarrow \epsilon \sim \frac{u_0^3}{\ell_0} \tag{1.7}$$

Se llega a una expresión de la disipación que es independiente de la viscosidad para Reynolds suficientemente grandes. Todavía quedan preguntas abiertas, como cuáles son las escalas más pequeñas de la turbulencia, o cómo se comportan el resto de escalas cuando decrece el tamaño de los remolinos. Para esto Kolmogorov hizo tres hipótesis.

- **Hipótesis de isotropía local:** *Para un número de Reynolds lo suficientemente grande, los movimientos turbulentos en escalas pequeñas $\ell \ll \ell_0$ son estadísticamente isótropos.*

  Según esta hipótesis, las escalas pequeñas se olvidan de las condiciones de los remolinos más grandes y se genera una escala universal que se comporta de manera similar para cualquier Reynolds alto. Lo más importante de esta hipótesis es que existe un momento en el que la turbulencia se olvida del origen. Si miramos en una caja suficientemente pequeña, no tenemos información del flujo principal, todo es isótropo. Estas escalas se definen por la siguiente condición.

  $$\ell < \ell_{EI} \tag{1.8}$$

  Donde:

  $$\ell_{EI} \approx \frac{1}{6}\ell_0 \tag{1.9}$$

- **Primera hipótesis de similitud de Kolmogorov:** *Para cualquier flujo turbulento a Reynolds suficientemente alto, las estadísticas de los movimientos a escalas pequeñas, $\ell < \ell_{EI}$, tienen una forma universal que dependen únicamente de ν y ϵ.*

Este es el rango que recibe el nombre de *Rango de Equilibrio Universal*, y las escalas temporales cumplen que:

$$\ell/u(\ell) \ll \ell_0/u_0 \tag{1.10}$$

Los remolinos dan vueltas más rápido, y como transfieren energía por vuelta, estos cada vez transfieren energía más rápido y viven menos. Por la hipótesis de isotropía local, en estas escalas no heredamos ni ℓ_0 ni u_0, por lo que las escalas vienen definidas por la disipación (ϵ) y la viscosidad (ν).

$$\begin{aligned} \eta &\equiv (\nu^3/\epsilon)^{1/4} \\ u_\eta &\equiv (\nu\epsilon)^{1/4} \\ \tau_\eta &\equiv (\nu/\epsilon)^{1/2} \end{aligned} \tag{1.11}$$

Y como cabría esperar, el número de Reynolds en estas escalas es:

$$Re_\eta = \frac{\eta u_\eta}{\nu} = 1 \tag{1.12}$$

Se llega a que las escalas inerciales y viscosas son del mismo orden, y la viscosidad es capaz de laminar los remolinos. Podemos hacer una comparación de las escalas integrales y las disipativas del flujo en función del número de Reynolds, obteniendo lo siguiente:

$$\begin{aligned} \frac{\eta}{\ell_0} &\sim Re^{-3/4} \\ \frac{u_\eta}{u_0} &\sim Re^{-1/4} \\ \frac{\tau_\eta}{\tau_0} &\sim Re^{-1/2} \end{aligned} \tag{1.13}$$

Como se puede apreciar, a mayor número de Reynolds, las escalas espaciales se hacen cada vez más pequeñas, complicando la resolución computacional de problemas turbulentos a altos números de Reynolds. Además, debe existir un rango de escalas ℓ tales que son muy pequeñas comparadas con ℓ_0, pero muy grandes comparadas con η. Estas escalas se encuentran entre $\ell_0 \gg \ell \gg \eta$, con un número de Reynolds lo suficientemente grande para no estar afectadas por la viscosidad. De aquí nace la siguiente hipótesis de Kolmogorov.

- **Hipótesis de similitud segunda de Kolmogorov:** *Para cualquier flujo turbulento a Reynolds suficientemente altos, las estadísticas de los movimientos a escalas comparables a ℓ en el rango $\ell_0 \gg \ell \gg \eta$, tienen una forma que viene determinada únicamente por la disipación y es independiente de la viscosidad.*

  Así nace una nueva escala:

$$\ell_{DI} \approx 60\eta \tag{1.14}$$

La cual nos divide el rango de equilibrio universal en dos, el subrango inercial y el disipativo. La característica más importante de este rango es que se produce una transferencia sucesiva de energía hacia escalas cada vez más pequeñas. En el subrango inercial no se puede usar la viscosidad para la construcción de las distintas escalas. Si tomamos un tamaño típico de remolino ℓ, las escalas las podemos definir como:

$$\begin{aligned} u(\ell) &= (\epsilon\ell)^{1/3} \sim u_0(\ell/\ell_0)^{1/3} \\ \tau(\ell) &= (\ell^2/\epsilon)^{1/3} \sim \tau_0(\ell/\ell_0)^{2/3} \end{aligned} \tag{1.15}$$

Como la tasa de transferencia de energía es $u(\ell)^2/\tau(\ell) = \epsilon$, obtenemos la tasa de disipación de energía. Con esto obtenemos que la tasa de transmisión es independiente de la escala ℓ. En el subrango inercial, toda la energía se transmite hacia la escala disipativa.

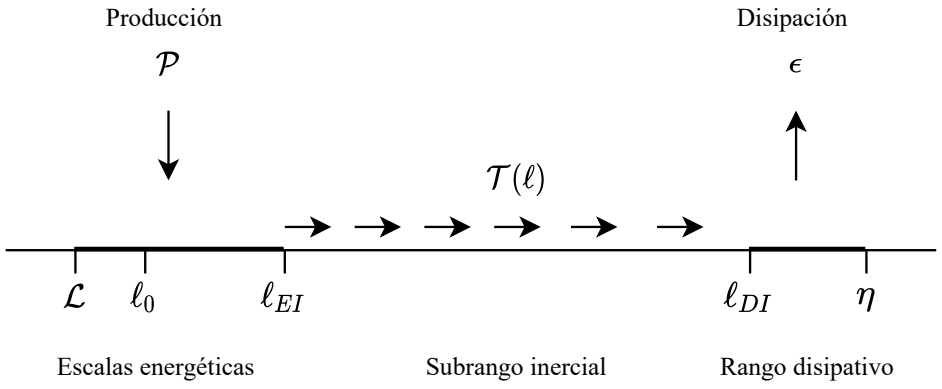


Figura 1.1: Representación de la transferencia de energía entre las distintas escalas de la turbulencia

Todas estas ideas se puede representar esquemáticamente en lo que se conoce como *Cascada de energía de Kolmogorov*, la cual podemos ver esquemáticamente en la figura 1.2. Se puede apreciar que la energía entra al flujo por las grandes escalas, y esta se trasmite hasta escalas más pequeñas donde los efectos viscosos son importantes y laminan el flujo [3].

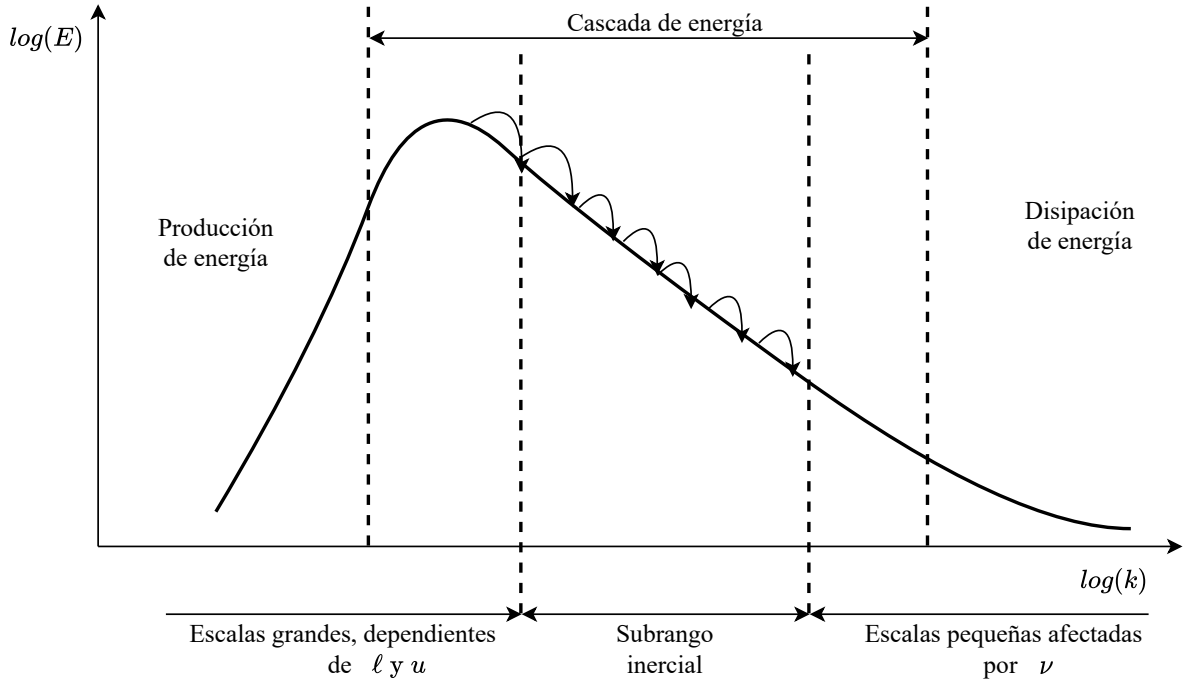


Figura 1.2: Representación esquemática de la cascada de energía de Kolmogorov

1.2. Flujos de pared

Uno de los casos más importantes desde el punto de vista ingenieril es el estudio de los flujos de pared. Su estudio ha sido muy importante a lo largo del siglo XX, sobre todo en el estudio de tuberías. La particularidad más importante de este tipo de flujos es que la presencia de la pared hace que se pierda la isotropía y homogeneidad del flujo, debido a que ahora el comportamiento del flujo si depende de la dirección en la que se trabaje.

La condición más importante de estos flujos es la condición de no deslizamiento, por la cual las partículas que se encuentran en contacto con la pared llevan la misma velocidad que esta. Además, esta condición es la responsable de la aparición de esfuerzos viscosos que se pueden escribir como:

$$\tau = \nu \frac{\partial U}{\partial y} \tag{1.16}$$

Existen tres casos típicos de turbulencia de pared:

- **Canal plano:** La geometría de este tipo de problemas consta de dos placas separadas una distancia $2h$ entre sí. Consideramos el *eje* x paralelo a la dirección del flujo,

el *eje y* perpendicular a las placas, y el *eje z* paralelo a las placas y perpendicular a la dirección del flujo (como se muestra en la figura 1.3). En este caso se crea una capa límite en ambas paredes y hace que todo el dominio este influenciado por la presencia de estas. Los planos paralelos a la pared se consideran infinitos, lo que nos permite usar esquemas periódicos espectrales en esas direcciones. En función de lo que produzca el movimiento del fluido, podemos diferenciar tres casos distintos: una diferencia de velocidades entre las placas (flujo de Couette), gradiente de presiones (flujo de Poiseuille), o una combinación entre ambas (flujo de Couette-Poiseuille). Para su estudio se considera que el flujo está estadísticamente desarrollado.

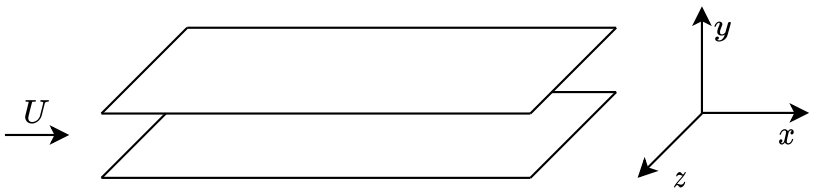


Figura 1.3: Representación esquemática del flujo en un canal

- **Tubería:** en este caso tenemos simetría radial, por lo que tenemos una dirección infinita y la otra periódica. Este tipo de simetrías es usual abordarlos en coordenadas cilíndricas. Es usual el estudio experimental con este tipo de flujo, siendo el experimento de Marusic *et al* [4] con $Re_\tau \sim 10^5$ de los más importantes en este ámbito.

Figura 1.4: Representación esquemática del flujo en una tubería

- **Capa límite:** En este caso tenemos la presencia de la pared por un lado, y por otro lado el flujo libre. Las capas límites se están desarrollando continuamente, por lo que las derivadas de las magnitudes estadísticas no llegan a ser 0. Esto hace que su resolución numérica sea mucho más compleja excepto en casos muy sencillos. Es usual considerar que el flujo es uniforme en dirección z, por lo que se puede simplificar en casos bidimensionales. En la figura 1.5 tenemos una representación esquemática del desarrollo del flujo en una capa límite.

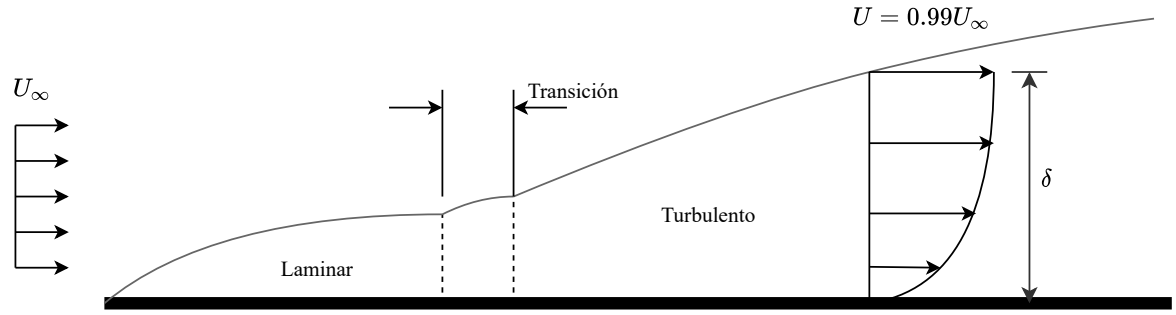


Figura 1.5: Representación esquemática del flujo en una capa límite

Este trabajo se centrará en el canal plano, debido a la eficiencia computacional de poder usar un método espectral. A partir de las ecuaciones de Navier-Stokes, y dividiendo las componentes de la velocidad en media y en perturbación (se profundizará más adelante), podemos obtener las ecuaciones medias del canal:

$$\frac{\partial \langle uv \rangle}{\partial y} = -\frac{1}{\rho}\frac{\partial \langle P \rangle}{\partial x} + \nu \frac{\partial^2 \langle U \rangle}{\partial y^2} \tag{1.17}$$

$$\frac{\partial \langle v^2 \rangle}{\partial y} = -\frac{1}{\rho}\frac{\partial \langle P \rangle}{\partial y} \tag{1.18}$$

A partir de la segunda ecuación, integrándola en la dirección y, podemos obtener una constante de integración en la pared con unidades de velocidad al cuadrado, la cual es muy útil para caracterizar flujos de pared. Esta magnitud recibe el nombre de velocidad de fricción (u_τ) o esfuerzo cortante en la pared, y se puede definir como:

$$u_\tau^2 = \nu \left.\frac{\partial U}{\partial y}\right|_w \tag{1.19}$$

Esta velocidad de fricción nos permite definir velocidades y distancias adimensionales, además de el número de Reynolds que mejor define este tipo de problemas:

$$U^+ = \frac{U}{u_\tau},\ y^+ = \frac{y u_\tau}{\nu},\ Re_\tau = \frac{u_\tau h}{\nu} \tag{1.20}$$

Esto nos lleva a una diferenciación de distintas capas en la capa límite del canal en función de la altura con respecto a la pared y^+. Por un lado, muy cerca de la pared, los esfuerzos viscosos son muy importantes, llegando a $\partial_{y^+} \langle U \rangle^+ = 1$, conocida como la subcapa viscosa. En esta región se cumple que:

$$u^+ = y^+ \tag{1.21}$$

Esta región se extiende desde la pared hasta un $y^+ < 5$ aproximadamente. Por el contrario, en el centro del canal, tenemos la condición de simetría, y por otro lado que el efecto viscoso es muy pequeño, por lo que $\partial_{y^+} \langle U \rangle^+ = 0$. Esta capa recibe el nombre de región exterior, cuya ecuación es la siguiente:

$$- \langle uv \rangle^+ = (1 - Y), \tag{1.22}$$

con $Y = y/h$. Esta región se localiza a partir de $Y > 0.2$. Von Kármán postulo que entre ambas capas debe existir una región logarítmica que se rige por la siguiente ecuación:

$$U^+ = \frac{1}{\kappa} \log y^+ + B, \tag{1.23}$$

donde κ es la constante de von Kármán y B es una constante. Ambas constantes se pueden estimar de manera experimental o mediante simulaciones DNS. Actualmente el valor de estas constantes es $\kappa = 0.388$ y $B = 5.1$. La capa logarítmica se extiende desde $y^+ > 100$ hasta $Y < 0.1 - 0.3$. Además, existe otra región entre la subcapa viscosa y la capa logarítmica, denominada capa *buffer*, en la que el flujo se adapta de una ley a otra.

1.3. Simulaciones numéricas de mecánica de fluidos

El objetivo de las simulaciones de mecánica de fluidos por ordenador es obtener resultados de ciertas variables de interés que tienen relevancia práctica. Resolver la turbulencia es un reto difícil, ya que estamos interesados en obtener $\boldsymbol{U}(\boldsymbol{x}, t)$ la cual es tridimensional, dependiente del tiempo y caótica. Como se ha visto en el apartado de la *Cascada de Kolmogorov*, la escala más pequeña del flujo, la escala de Kolmogorov η, escala con el Reynolds de la forma $Re^{-3/4}$. Esto implica que para Reynolds medianamente altos, sea imposible resolver todas las escalas de la turbulencia. Todo esto nos lleva a que existan tres aproximaciones distintas en función del objetivo de la simulación: RANS, LES y DNS.

1.3.1. Reynolds Averaged Navier-Stokes

Este método, también denominado RANS, no resuelve ninguna escala de la turbulencia. Se realiza un cálculo de flujo medio y se modelan los efectos de la turbulencia. Para ello, se han desarrollado diversos modelos, dependiendo del tipo de problema en el que nos encontremos, con constantes que deben ser ajustadas experimentalmente. Debido a que la turbulencia no se resuelve, el coste computacional de este tipo de problemas es muy bajo y por lo tanto es ampliamente usado en la industria.

Para obtener las ecuaciones de este tipo de simulaciones, empezamos por separar magnitudes físicas en su parte homogénea y su parte turbulenta de la siguiente forma:

$$U_i = \langle U_i \rangle + u_i \tag{1.24}$$

Siendo la velocidad total y la perturbación dependientes del espacio y del tiempo, y la media solo depende de la posición espacial. Aplicando esta descomposición, podemos llegar a la siguiente expresión de las ecuaciones de Navier-Stokes (de forma incompresible).

$$\frac{\partial \langle U_j \rangle}{\partial} + \langle U_i \rangle \frac{\partial}{\partial x_i} \langle U_j \rangle = \nu \nabla^2 \langle U_j \rangle - \frac{1}{\rho} \frac{\partial \langle P \rangle}{\partial x_j} - \frac{\partial}{\partial x_i} \langle u_i u_j \rangle \tag{1.25}$$

Como podemos ver, aparecen los mismos términos que las ecuaciones de Navier-Stokes originales salvo el último término, que recibe el nombre de tensor de Reynolds o Reynold stress. Este termino representa la transferencia de momento por las fluctuaciones de la velocidad. Analizando el problema, tenemos en total 4 ecuaciones (continuidad más tres ecuaciones de Navier-Stokes), y como incógnitas tenemos las 3 componentes de la velocidad medias, la presión y el tensor de Reynolds. Este tensor tiene la siguiente forma.

$$R_{ij} = \langle u_i u_j \rangle = - \begin{pmatrix} \langle u^2 \rangle & \langle uv \rangle & \langle uw \rangle \\ \langle uv \rangle & \langle v^2 \rangle & \langle vw \rangle \\ \langle uw \rangle & \langle vw \rangle & \langle w^2 \rangle \end{pmatrix} \tag{1.26}$$

Este tensor añade 6 incógnitas más al problema. Esto es lo que se conoce como problema de cierre. Para *cerrar* el problema lo que se hace es aplicar modelos que nos permiten estimar los esfuerzos de Reynolds. Así, al resolver RANS únicamente tenemos que resolver el flujo medio en la geometría deseada, sin resolver directamente la turbulencia. Esto es lo que permite que sean tan baratos computacionalmente. Existen dos tipos de modelos de cierre: a través de la hipótesis de viscosidad turbulenta, o directamente el modelado de las ecuaciones de transporte para los esfuerzos de Reynolds.

Como hemos dicho, la forma más habitual de cerrar estos problemas es a partir de la hipótesis de viscosidad turbulenta (hipótesis de Boussinesq). En 1877, Bussinesq postuló en [5] que el tensor de esfuerzos de Reynolds es proporcional al tensor de tasa de deformación media anisótropo. Esto puede ser escrito como:

$$\langle u_i u_j \rangle = \frac{2}{3} k \delta_{ij} - \nu_T \left(\frac{\partial \langle U_i \rangle}{\partial x_j} + \frac{\partial \langle U_j \rangle}{\partial x_i} \right\rangle, \tag{1.27}$$

donde $k = 1/2 \langle u_i u_i \rangle$ es la energía cinética turbulenta y δ_{ij} es la delta de Kronecker. Los modelos que usan esta aproximación reciben el nombre de *Eddy Viscosity Models* y son

ampliamente utilizados en la industria ya que basta con modelar la viscosidad turbulenta y la energía cinética turbulenta, lo que reduce el coste computacional. Existen modelos más complejos que utilizan ecuaciones de transporte adicionales que permite mejorar la precisión de los mismos. Sin embargo, ningún modelo se escapa del uso de constantes que necesitan ser calibradas para cada caso concreto. Estas constantes se puede obtener experimentalmente o mediante cálculos DNS o LES. Entre los modelos usados destacan: el modelo Spalart-Allmaras, el cual modela la viscosidad turbulenta con una única ecuación, dando resultados razonables para problemas de aerodinámica externa; el modelo $k-\epsilon$, el cual utiliza una ecuación de desarrollo riguroso para la k, mientras que para ϵ se hace uso de una aproximación empírica, es uno de los modelos más comunes en los problemas RANS, con un rango de aplicabilidad muy amplio aunque falla en casos con gradientes de presión adversos importantes o flujos muy rotaciones; el modelo $k-\omega$, el cual usa una ecuación de transporte para la disipación turbulenta específica $\omega = \epsilon/k$, funciona mejor que el modelo anterior para regiones cercanas a la pared, pero falla en zonas de flujo libre; y finalmente el modelo $k-\omega$ SST el cual utiliza modelos $k-\omega$ cerca de la pared y transiciona a modelos $k-\epsilon$ en el flujo libre, dando resultados razonables en todos los casos [3].

Por otro lado tenemos los modelos que se encargan de modelar el tensor de esfuerzos de Reynolds (*Reynolds Stress Equation*), en la que existe una ecuación de transporte por cada uno de los seis términos del tensor. Esto mejora la precisión obtenida respecto a los modelos EVM ya que se evita el uso de una viscosidad turbulenta. Además, se suele usar una ecuación de transporte adicional para modelar la disipación ϵ. El hecho de resolver 7 ecuaciones de transporte hace que aumente demasiado el coste computacional, cuando seguimos sin resolver ninguna escala de la turbulencia, por lo que estos modelos son menos usados que los basados en la viscosidad turbulenta.

Finalmente, a pesar que estos métodos tienen sus pros y sus contras, siguen teniendo el mismo problema: no resolvemos ninguna escala de la turbulencia, y las escalas inerciales no son universales y dependen del tipo de problema, por lo que los cálculos RANS deben ser calibrados mediante constantes para obtener resultados útiles. Además, un paso adicional antes de poder usar los resultados obtenidos es la verificación de los mismos, ya sea a través de experimentos en túnel de viento o bibliográficos.

1.3.2. Large Eddy Simulation

Los cálculos de Large-Eddy Simulation (LES) nos permiten resolver problemas turbulentos no estacionarios, resolviendo las escalas grandes del flujo y modelando las más pequeñas. En términos de coste computacional, estos modelos se encuentran entre RANS y DNS, y nacieron a partir de las limitaciones que tienen estos dos últimos. Por un lado nos permite obtener resultados más precisos y fiables que los modelos *Reynolds Stress Equations*, sobre todo en problemas de separación no estacionaria, o desprendimiento de vórtices. Por otro lado, al evitar resolver las escalas más pequeñas de la turbulencia, podemos usar mallas mucho más gruesas que en las DNS, lo que permite tiempos de cálculo razonables. El

mayor coste computacional en los cálculos DNS se invierte en resolver la escala disipativa, mientras que la mayor parte de la energía se encuentra en las escalas integrales [3].

La forma de abordar estos problemas es mediante el filtrado de las ecuaciones de Navier-Stokes de los números de onda más grandes. Ahora el término no lineal de los esfuerzos de Reynolds tiene que tener en cuenta las escalas no resueltas: las escalas que se encuentran por debajo del filtro (SFS, *sub-filter scale*) y las escalas más pequeñas que el tamaño de la malla (SGS, *sub-grid scale*). Existen cuatro pasos conceptuales que se siguen a la hora de resolver un caso en LES:

- Se utiliza una operación de filtrado que descompone $\boldsymbol{U}(\boldsymbol{x}, t)$ en una componente filtrada (o resuelta) $\overline{\boldsymbol{U}}(\boldsymbol{x}, t)$ y una componente residual (SGS) $\boldsymbol{u}'(\boldsymbol{x}, t)$. El término $\overline{\boldsymbol{U}}(\boldsymbol{x}, t)$ representa el movimiento de las escalas más grandes del fluido.

- Obtenemos las ecuaciones para el campo de velocidades filtradas a partir de las ecuaciones de Navier-Stokes. En este caso aparece un término que es el tensor de esfuerzos de Reynolds residuales (o *SGS stress tensor*).

- El cierre del problema viene a partir de modelar este tensor de esfuerzos residuales, normalmente a partir de el uso de una viscosidad turbulenta, de forma análoga a lo realizado en RANS.

- Resolvemos el modelo con las ecuaciones filtradas, que nos da los resultados para $\overline{\boldsymbol{U}}(\boldsymbol{x}, t)$, lo que nos proporciona una aproximación a los movimientos de las escalas más grandes [3].

El primero modelo utilizado para modelar la SGS (y el más famoso) es el de Smagorinsky-Lilly [6], el cual modela la viscosidad turbulenta de la siguiente forma:

$$\nu_T = (C_S \Delta_x)^2 \sqrt{\frac{1}{2}\left(\frac{\partial \overline{U_i}}{\partial x_j} + \frac{\partial \overline{U_j}}{\partial x_i}\right)\left(\frac{\partial \overline{U_i}}{\partial x_j} + \frac{\partial \overline{U_j}}{\partial x_i}\right)} \tag{1.28}$$

Donde Δ_x es el tamaño de la malla y C_S es el coeficiente de Smagorinsky. La disipación viene dada por el tamaño del filtro. Los modelos LES resuelven aproximadamente el 80 % de la energía de la cascada.

Los modelos LES tienen un coste computacional superior a los cálculos RANS, por lo que su aplicación de forma industrial, a día de hoy, sigue siendo limitada. Los modelos LES nacieron debido a que hay aplicaciones cuyas escalas grandes tiene gran relevancia, como por ejemplo aplicaciones meteorológicas y capas límite atmosféricas, y resolver las escalas grandes son fundamentales.

1.3.3. Direct Numerical Simulation

En este caso se resuelven todas las escalas de la turbulencia, partiendo únicamente de las ecuaciones de Navier-Stokes, las condiciones iniciales y las condiciones de contorno y sin hacer ninguna suposición extra. Esto hacía que su resolución fuera inabarcable hasta los años 70s, cuando el aumento de la potencia de cálculo permitió resolver las primeras DNS. El incremento del coste computacional es muy rápido con el aumento del número de Reynolds. Tal y como muestra Jiménez en [7], en el caso de turbulencia anisótropa (como es el caso de los canales), el escalado con el número de Reynolds es todavía superior, ya que la anisotropía requiere cajas computacionales mayores. Se estima que el número de operaciones escala como $Re_\tau^3 Re_\tau^{3/4} = Re_\tau^{15/4}$.

La implantación de estos algoritmos normalmente requieren geometrías sencillas. Se utilizan cajas con mallas estructuradas cartesianas. Esto hace que su aplicación en el ámbito industrial sea prácticamente nulo y se ciña al ámbito científico.

Las primeras aplicaciones de las DNS fueron sobre turbulencia homogénea. En estos problemas tenemos que no hay distinción entre las 3 direcciones del fluido, por lo que podemos aplicar periodicidad en cada una de ellas. El hecho de aplicar periodicidad nos permite usar una descomposición en series de Fourier,

$$\boldsymbol{U}(\boldsymbol{x}, t) = \sum_{\kappa} e^{i\kappa x} \hat{\boldsymbol{U}}(\boldsymbol{\kappa}, t), \tag{1.29}$$

donde κ es el número de onda, y hay un total de N^3 números de onda (para cada dirección). Esto permite simplificar la resolución de las ecuaciones de Navier-Stokes, ya que operaciones como la derivada o la integral se convierten en multiplicaciones y divisiones. Además, para cambiar entre el dominio de la frecuencia y el espacio físico se puede hacer uso de la transformada de Fourier rápida (FFT), la cual tiene un coste de $N^3 \log N$ operaciones. Esto es lo que se conoce como métodos espectrales. En el caso de las ecuaciones de Navier-Stokes, cuenta con un termino no lineal, que al trabajar con él en el dominio de la frecuencia hace que las interacciones entre ellos aumenten las operaciones necesarias a N^6. Para evitar este problema, se hace una transformación al espacio físico para operar el término no lineal, y después se vuelve al espacio de Fourier. Esto es lo que se conoce como método pseudo-espectral, y tiene un coste computacional de $N^3 \log N$, aunque requieren de métodos de antialiasing para reducir el error. El primero en enfrentarse a todos estos problemas y sentó las bases de los algoritmos de DNS posteriores es Rogallo [8], el cual obtuvo sus primeros resultados numéricos en 1981.

En el caso de los flujos no homogéneos, como por ejemplo en el flujo en un canal, no podemos aplicar Fourier en la dirección no homogénea ya que necesitamos de condiciones de contorno (en este caso no podemos aplicar periodicidad) y el flujo cercano a la pared, caracterizado por la longitud viscosa δ_y, requiere un tratamiento especial para su resolución. En el caso del canal plano, tenemos una caja con dimensiones $\mathcal{L}_x \times h \times \mathcal{L}_z$, siendo las

direcciones x y z de flujo estadísticamente homogéneo y resueltas por series de Fourier, mientras que la dirección y suele ser resuelta mediante polinomios de Chebyshev.

La primera simulación de flujo anisótropo es la realizada por *Kim et al* [9] en el año 1987, siendo una simulación muy compleja por el alto coste computacional que suponía para la época. La evolución de las simulaciones DNS ha ido de la mano con el desarrollo de los diferentes supercomputadores. Tal y como indica Jiménez en [7], la potencia computacional aumenta por 1000 cada 15 años. Esto hace que la simulación de Kim se pueda resolver actualmente en ordenadores personales en cuestión de horas. La mayor simulación realizada hasta el momento de un canal turbulento es de un Reynolds de fricción de $Re_\tau = 10000$ [10] lo que supone que ya se van aproximando a los experimentalistas.

El cálculo computacional tiene como principal ventaja respecto a los experimentos la de poder obtener el valor del campo fluido en cada punto del dominio. Esto permite obtener estadísticas mucho más completas y un análisis mucho más exhaustivo. Por otro lado, el hecho de usar dominios computacionales tan grandes lleva a necesidades de memoria muy grandes, que pueden andar del orden de Petabytes [7]. Esto plantea dificultades técnicas para el manejo de los datos y la transferencia de datos entre distintos equipos.

1.4. Estructuras coherentes

Aunque la teoría de Kolmogorov es un gran pilar en el entendimiento de la turbulencia tal y como la conocemos ahora, presenta una serie de inconvenientes. Por un lado, Kolmogorov asumía que los gradientes de velocidad estaban uniformemente distribuidos a lo largo del espacio, pero experimentos como [11] en 1949 observaban que esto no era del todo cierto, ya que existía una intermitencia en el flujo. Experimentos y visualizaciones del flujo posteriores, como *Kline et al* [12], supusieron una forma totalmente distinta a cómo se veía la turbulencia en la época, ya que pudieron observar regiones coherentes dentro del flujo, a diferencia de las teorías estadísticas de Kolmogorov. Estos experimentos identificaron estructuras coherentes espacial y temporalmente en flujos de mezcla y capas límite, y pronto estas estructuras se convirtieron en una nueva manera de explicar los flujos turbulentos.

La definición de vórtice es compleja y no existe una definición con la que esté de acuerdo la comunidad científica. Saffman definió en su libro [13] un vórtice como *región finita de volumen rotacional dentro de un flujo irrotacional*. Esta definición es poco satisfactoria ya que lleva intrínseca la existencia de límites definidos entre vórtice y fluido, pero en los flujos reales estas fronteras definidas no existen, ya que la viscosidad difunde la vorticidad [14].

En términos generales, existen en el momento dos modelos que intentan explicar las estructuras las estructuras coherentes, aunque incompletos. En primer lugar tenemos el paradigma de los paquetes de los denominados *hairpins*, propuesto por Adrian, Meinhart

y Tomkins [15], los cuales son vórtices en forma de herradura que nacen cerca de la pared y van hacia la región exterior, con un tiempo de vida mayor a su tiempo de vida característico. El segundo modelo, más completo que el anterior, propuesto por del Álamo *et al* [16] en el que el flujo dentro de la capa logarítmica es explicado en términos de *sweeps* y *ejections*. Estos nacen a partir de una *streak* del flujo, la cual es una región fina y alargada (en la dirección del fluido) que se corresponde a un movimiento relativamente lento del fluido y cercano a la pared ($y^+ < 10$). Estos van creciendo lentamente en velocidad y llega un momento en el que se aleja de la pared muy rápidamente, generando los *ejections*. Este proceso se conoce *bursting*. Por continuidad, si hay parte del fluido que se aleja de la pared, otra parte del fluido debe ir hacia esta región. Estas estructuras de alta velocidad hacia la pared se denominan *sweeps*, y suelen agruparse en pares con los *ejections*.

1.5. Objetivo del trabajo

El objetivo fundamental de este trabajo es el desarrollo de herramientas que nos permitan identificar y seguir temporalmente estructuras coherentes en un canal turbulento resuelto mediante DNS, aplicando paralelización de memoria distribuida (MPI) y memoria compartida (OpenMP), con el objetivo de implementarlo en supercomputadores. Al ser un código paralelo se puede emplear para el número de Reynolds Re_τ que se desee, ya que va a ser perfectamente escalable con el tamaño del problema.

El punto de partida de este proyecto es el código DNS que identifica si cualquier punto del dominio supera o no cierto umbral para ser considerado vórtice. A partir de aquí, los objetivos de este trabajo son:

- **Identificación de estructuras en una matriz booleana** paralelizado de forma eficiente tanto en memoria como en potencia computacional.
- **Seguimiento temporal de los vórtices**, identificando la vida, muerte, rotura y coalescencia de las estructuras.
- **Implementación de los algoritmos al código DNS**, lo que nos permitirá calcular la evolución de los vórtices mientras se resuelve el flujo.
- **Programación de códigos MATLAB** que servirán para ayudar al desarrollo conceptual de los algoritmos que posteriormente se implementarán en Fortran, y programas que nos permitan obtener estadísticas.

Además, este trabajo consta con un apartado en el que se explicarán en detalle los métodos numéricos usados en la DNS y los métodos que nos permiten obtener los puntos vórtices del dominio. Finalmente, se hará una conclusión con los resultados más relevantes y un pliego de condiciones y presupuesto.

Capítulo 2

Simulaciones DNS

La simulación numérica directa, DNS, se encarga de resolver todas las escalas de la turbulencia, proporcionando gran cantidad de información en todo el campo fluido, pudiendo obtener estadísticas de forma más sencilla que en el caso de simulaciones experimentales. Como contrapartida, estas simulaciones son muy caras tanto por coste computacional como por memoria, lo que impide que su uso esté extendido en en la industria, y su uso se ciña en aplicaciones científicas a bajos números de Reynolds y geometrías muy sencillas.

La geometría del dominio es la mostrada en la figura 1.3 que consta de dos placas separadas una distancia $2h$. Las direcciones paralelas a la pared tienen condición de contorno de periodicidad, lo que nos permite usar métodos espectrales basados en la transformada discreta de Fourier, reduciendo así el coste computacional. La dirección perpendicular a la pared no tiene condiciones de contorno periódicas, por lo que no es posible el uso de series de Fourier y hay que recurrir al uso de diferencias finitas compactas de alta precisión en esta dirección.

El código DNS ya ha sido usado en múltiples trabajos como [17] [18] [19], por lo que consideramos que el código ya se encuentra validado y no es necesario hacerlo en este trabajo.

2.1. Método numérico

El método numérico empleado en estas simulaciones es el que desarrollaron Kim, Moin y Moser en [9]. Si definimos el campo de velocidades como $\boldsymbol{U} = (u, v, w)$ y el campo de vorticidades como $\boldsymbol{\Omega} = (\omega_x, \omega_y, \omega_z)$ (siendo $\boldsymbol{\Omega} = \nabla \times \boldsymbol{U}$), el método numérico se basa en expresar las ecuaciones de Navier-Stokes en función del laplaciano de la velocidad normal a la pared $\phi = \Delta v$ y la vorticidad normal a la pared ω_y. El punto de partida para esta demostración son las ecuaciones de Navier-Stokes y la ecuación de continuidad para un flujo incompresible:

$$\nabla \boldsymbol{U} = 0 \tag{2.1}$$

$$\frac{\partial \boldsymbol{U}}{\partial t} + \boldsymbol{U} \cdot \nabla \boldsymbol{U} = -\frac{1}{\rho}\nabla P + \nu \Delta \boldsymbol{U} \tag{2.2}$$

Introducimos una nueva magnitud $\boldsymbol{H}$, llamada *helicidad*, la cual mide en cada punto la proyección de la vorticidad sobre la velocidad, por lo que su definición es la siguiente:

$$\boldsymbol{H} = \boldsymbol{U} \times \boldsymbol{\Omega} \tag{2.3}$$

Podemos desarrollar el término convectivo de la ecuación (2.2) aplicando la siguiente propiedad del gradiente:

$$\nabla(\boldsymbol{A} \cdot \boldsymbol{B}) = \boldsymbol{B} \times (\nabla \times \boldsymbol{A}) + \boldsymbol{A} \times (\nabla \times \boldsymbol{B}) + (\boldsymbol{B} \cdot \nabla)\boldsymbol{A} + (\boldsymbol{A} \cdot \nabla)\boldsymbol{B} \tag{2.4}$$

Aplicando la propiedad de la ecuación (2.4) sobre el término $\boldsymbol{U} \cdot \nabla \boldsymbol{U}$ podemos obtener la siguiente expresión:

$$\frac{1}{2}\nabla(\boldsymbol{U} \cdot \boldsymbol{U}) = \boldsymbol{U} \times (\nabla \times \boldsymbol{U}) + \boldsymbol{U} \cdot \nabla \boldsymbol{U} = \boldsymbol{H} + \boldsymbol{U} \cdot \nabla \boldsymbol{U} \tag{2.5}$$

Añadiendo este resultado a la ecuación (2.2) y adimensionalizando la viscosidad se obtiene:

$$\frac{\partial \boldsymbol{U}}{\partial t} + \frac{1}{2}\nabla(\boldsymbol{U} \cdot \boldsymbol{U}) - \boldsymbol{H} = -\frac{1}{\rho}\nabla P + \frac{1}{Re_c}\Delta \boldsymbol{U} \tag{2.6}$$

Donde $Re_c = u_c h/\nu$ es el número de Reynolds basado en la velocidad media del canal u_c, que viene definida como:

$$u_c = \frac{1}{2h}\int_{-h}^{h} \langle U \rangle \, dy \tag{2.7}$$

Una vez tenemos las ecuaciones de Navier-Stokes expresadas en función de la helicidad, el siguiente paso es aplicar la divergencia a la ecuación (2.6) para obtener la ecuación de transporte de la divergencia de la helicidad $\nabla \boldsymbol{H}$. Aplicamos además la propiedad conmutativa que tiene la divergencia con la derivada temporal y con el laplaciano:

$$\frac{\partial(\nabla \boldsymbol{U})}{\partial t} + \frac{1}{2}\Delta(\boldsymbol{U}\cdot\boldsymbol{U}) - \nabla \boldsymbol{H} = -\frac{1}{\rho}\Delta P + \frac{1}{Re_c}\Delta(\nabla \boldsymbol{U}) \tag{2.8}$$

Que se puede simplificar aplicando la continuidad (2.2) obteniendo la siguiente expresión:

$$\nabla \boldsymbol{H} = \frac{1}{2}\Delta(\boldsymbol{U}\cdot\boldsymbol{U}) + \frac{1}{\rho}\Delta P \tag{2.9}$$

Por otro lado, podemos aplicar el laplaciano a la ecuación (2.6):

$$\frac{\partial(\Delta \boldsymbol{U})}{\partial t} + \nabla\left(\frac{1}{2}\Delta(\boldsymbol{U}\cdot\boldsymbol{U})\right) - \Delta \boldsymbol{H} = -\nabla\left(\frac{1}{\rho}\Delta P\right) + \frac{1}{Re_c}\Delta^2 \boldsymbol{U} \tag{2.10}$$

Y podemos introducir el resultado obtenido en la ecuación (2.9), por lo que se puede simplificar a:

$$\frac{\partial(\Delta \boldsymbol{U})}{\partial t} + \nabla(\nabla \boldsymbol{H}) - \Delta \boldsymbol{H} = \frac{1}{Re_c}\Delta^2 \boldsymbol{U} \tag{2.11}$$

Ya hemos obtenido una de las expresiones finales de las ecuaciones de Navier-Stokes en función del laplaciano de la velocidad y la helicidad. La siguiente ecuación se obtiene aplicando el operador rotacional a la ecuación 2.6:

$$\frac{\partial \boldsymbol{\Omega}}{\partial t} + \frac{1}{2}\nabla \times \nabla(\boldsymbol{U}\cdot\boldsymbol{U}) - \nabla \times \boldsymbol{H} = -\frac{1}{\rho}\nabla \times \nabla P + \frac{1}{Re_c}\Delta \boldsymbol{\Omega} \tag{2.12}$$

Si aplicamos la propiedad de que $\nabla \times \nabla = 0$, la ecuación anterior se puede simplificar hasta obtener:

$$\frac{\partial \boldsymbol{\Omega}}{\partial t} - \nabla \times \boldsymbol{H} = \frac{1}{Re_c}\Delta \boldsymbol{\Omega} \tag{2.13}$$

Hemos obtenido las ecuaciones (2.11) y (2.13) que son ecuaciones que se aplican en las tres direcciones de la caja, pero en este caso tomaremos la ecuación en la dirección perpendicular a la pared, con lo que podemos obtener las ecuaciones finales que se implementarán en el código DNS,

$$\frac{\partial \phi}{\partial t} = h_v + \frac{1}{Re_c}\Delta\phi, \tag{2.14}$$

$$\frac{\partial \omega_y}{\partial t} = h_g + \frac{1}{Re_b}\Delta\omega_y, \tag{2.15}$$

donde h_v y h_g se pueden expresar como,

$$h_v = \left(\frac{\partial^2}{\partial x^2} + \frac{\partial^2}{\partial z^2}\right) H_2 - \frac{\partial}{\partial y}\left(\frac{\partial H_1}{\partial x} + \frac{\partial H_3}{\partial z}\right), \tag{2.16}$$

$$h_g = \frac{\partial H_1}{\partial z} - \frac{\partial H_3}{\partial x}, \tag{2.17}$$

donde H_1, H_2 y H_3 son las tres componentes del vector helicidad. Finalmente, para cerrar el problema se hace uso de la definición de vorticidad y aplicando la ecuación de continuidad.

$$\frac{\partial u}{\partial x} + \frac{\partial v}{\partial y} + \frac{\partial w}{\partial z} = 0 \tag{2.18}$$

$$\frac{\partial u}{\partial z} - \frac{\partial w}{\partial x} = \omega_y \tag{2.19}$$

$$\frac{\partial \omega_x}{\partial x} + \frac{\partial \omega_y}{\partial y} + \frac{\partial \omega_z}{\partial z} = 0 \tag{2.20}$$

$$\frac{\partial \omega_z}{\partial x} - \frac{\partial \omega_x}{\partial z} = \phi \tag{2.21}$$

Una de las ventajas que presenta realizar esta transformación es que no necesitamos calcular la presión para obtener la solución al sistema, por lo que se puede calcular a posteriori para obtener estadísticas turbulentas que requieran su uso. Otra de las ventajas es que hemos pasado de 3 ecuaciones de transporte, una para cada componente de la velocidad, a dos ecuaciones, lo que supone una reducción del coste computacional. Una vez tenemos v y ω_y, la velocidad en la dirección del flujo u y la velocidad transversal w se puede obtener fácilmente a través de la ecuación de continuidad (2.18) y la definición de vorticidad (2.19), ya que las derivadas e integrales en x y z son triviales en el espacio de Fourier.

Como integrador temporal se ha usado un Runge-Kutta de tercer orden presentado en [20] el cual plantea una ecuación general del siguiente tipo.

$$\frac{\partial \boldsymbol{v}}{\partial t} = L(\boldsymbol{v}) + N(\boldsymbol{v}) \tag{2.22}$$

Donde $\boldsymbol{v}$ es un campo cualquiera, L es el operador lineal de la ecuación y N es el operador no lineal. En el caso de las ecuaciones de Navier-Stokes, el operador lineal sería el termino viscoso y el termino de presión, y el operador no lineal es el término convectivo. Así, para pasar de $\boldsymbol{v}_n$, en un instante temporal t, a $\boldsymbol{v}_{n+1}$, en el instante $t + \Delta t$, son necesarios 3 pasos:

$$\begin{aligned} \boldsymbol{v}' &= \boldsymbol{v}_n + \Delta t \left[L(\alpha_1 \boldsymbol{v}_n + \beta_1 \boldsymbol{v}') + \gamma_1 N_n \right] && (2.23) \\ \boldsymbol{v}'' &= \boldsymbol{v}' + \Delta t \left[L(\alpha_2 \boldsymbol{v}' + \beta_2 \boldsymbol{v}'') + \gamma_2 N' + \xi_1 N_n \right] && (2.24) \\ \boldsymbol{v}_{n+1} &= \boldsymbol{v}'' + \Delta t \left[L(\alpha_3 \boldsymbol{v}'' + \beta_3 \boldsymbol{v}_{n+1}) + \gamma_3 N'' + \xi_2 N' \right] && (2.25) \end{aligned}$$

Donde:

$$\begin{aligned} \gamma_1 &= \frac{8}{15}, \quad \gamma_2 = \frac{5}{12}, \quad \gamma_3 = \frac{3}{4}, \quad \xi_1 = -\frac{17}{60}, \quad \xi_2 = -\frac{5}{12} \\ \alpha_1 &= \frac{29}{96}, \quad \alpha_2 = -\frac{3}{40}, \quad \alpha_3 = \frac{1}{6} \\ \beta_1 &= \frac{37}{160}, \quad \beta_2 = \frac{5}{24}, \quad \beta_3 = \frac{1}{6} \end{aligned}$$

Las ventajas de este integrador temporal es que la memoria que requiere es mínima, del mismo orden que el método de Euler explícito, tiene una mayor estabilidad y un mayor paso temporal. Sin embargo, la desventaja es que necesitamos calcular tres operadores implícitos, lo que requiere más coste computacional.

Una vez descrito todo el método faltan imponer las condiciones de contorno: la condición de no deslizamiento $v|_{y=\pm h} = 0$, la velocidad en la pared $y|_{y=h}$ y el gradiente de presión $\partial P/\partial x$.

Al tratarse de un flujo no homogéneo, se resolverá usando series de Fourier en las direcciones paralelas a la pared con dealiasing, ya que este error afecta primero a las grandes escalas del flujo y rápidamente vuelve inestables los resultados a grandes números de onda, lo que obligaría a usar un atenuante de inestabilidad, cambiando por completo el espectro de transferencia de energía. En la dirección perpendicular a la pared se hará uso de diferencias finitas compactas de siete puntos con resolución espectral [21], que nos permite resolver con buena precisión y nos permite colocar los puntos de la malla arbitrariamente, algo importante a la hora de captar correctamente la capa límite.

Finalmente, cabe destacar que los resultados son válidos cuando se ha llegado a un estado estadísticamente estacionario, por lo que deberemos esperar a que el campo converja para poder obtener resultados.

2.2. Criterios para la identificación de estructuras

En la sección §1.4 se ha visto que la forma de estudiar la turbulencia actualmente pasa por el estudio de regiones coherentes dentro del flujo. Sin embargo, estas regiones coherentes no son fáciles de definir ya que implican la existencia de una frontera entre regiones vorticales y zonas irrotacionales. Esta frontera estricta no existe en los flujos reales debido a que la viscosidad difunde la vorticidad. Entre los problemas existentes a la definición de vórtice encontramos los siguientes [14]:

- Los flujos viscosos difunden el límite entre lo que consideramos vórtice y flujo exterior. Es por esto que no existe una verdad absoluta en el problema de identificación de los vórtices
- Puede ocurrir que varios núcleos de vórtice que acercan mucho y pueden solaparse. En este caso está la duda de si se considera un único vórtice o son dos separados.
- Definir un criterio de identificación a partir de principios físicos objetivos no es trivial.
- Existe discusión si son más efectivos los métodos Eulerianos contra los conceptos Lagrangianos.
- No existe consenso por parte de la comunidad científica en que método de identificación es el más efectivo, por lo que su elección sigue siendo subjetiva.

Actualmente existen multitud de criterios para la identificación de vórtices, que se puede clasificar como Eulerianos o Lagrangianos. Los Eulerianos son usados en simulaciones DNS por usar magnitudes Eulerianas, como por ejemplo los gradientes de velocidad. Los Lagrangianos usan los métodos Lagrangianos que se basan en el estudio de las líneas de corriente, ampliamente usados en experimentos. Dentro de los métodos Eulerianos se puede clasificar como locales (evaluación punto a punto) o no locales (dependen de múltiples puntos del dominio). Además, dentro de los métodos Eulerianos tenemos los métodos *region-type*, en los que una función escalar debe ser mayor (o menor) que un cierto umbral, y los métodos *line-type* que se encargan de buscar las líneas del esqueleto del vórtice.

A pesar de la existencia de múltiples criterios para la identificación de vórtices, si los criterios están bien definidos y los valores umbrales están bien elegidos, los resultados obtenidos son similares pero con discrepancias referentes al modelo escogido. Muchos de los criterios de vorticidad se basan en el tensor del vector velocidad.

$$A = \nabla u = \begin{bmatrix} \mathrm{U}_x & \mathrm{U}_y & \mathrm{U}_z \\ \mathrm{V}_x & \mathrm{V}_y & \mathrm{V}_z \\ \mathrm{W}_x & \mathrm{W}_y & \mathrm{W}_z \end{bmatrix} \tag{2.26}$$

Donde podemos definir la parte simétrica y la parte antisimétrica como:

$$S = \frac{1}{2}(\nabla u + \nabla u^T) \qquad \Omega = \frac{1}{2}(\nabla u - \nabla u^T) \tag{2.27}$$

Podemos buscar los autovalores del tensor 2.26 a partir de la ecuación:

$$\det[\nabla u - \lambda] = I \tag{2.28}$$

Lo que da lugar a una ecuación de la forma $\lambda^3 + P\lambda^2 + Q\lambda + R = 0$ de la que podemos obtener los siguientes tres invariantes.

$$P = \mathrm{tr}[A] = A_{ii} \tag{2.29}$$
$$Q = \frac{1}{2}[P^2 - S_{ij}S_{ji} - \Omega_{ij}\Omega_{ji}] \tag{2.30}$$
$$R = \frac{1}{3}[-P^3 + 3PQ - S_{ij}S_{jk}S_{ki} - 3\Omega_{ij}\Omega_{jk}S_{ki}] \tag{2.31}$$

Para los flujos incompresibles $P = 0$ por continuidad, por lo que el flujo se puede caracterizar a partir de los invariantes Q y R. La ecuación característica puede tener todos los autovalores reales y distintos, todos reales pero al menos dos son iguales, o un autovalor real y un par de complejos conjugados [22]. Los autovalores complejos son los que proporcionan el giro del flujo, por lo que gran parte de los criterios de identificación se basan en que existan autovalores complejos.

A pesar de que existen gran número de métodos para identificar vórtices, nosotros nos centraremos en dos: criterio de Chong con umbral no homogéneo y basado en eventos *uv* o también llamado identificación de Qs.

2.2.1. Criterio de Chong con umbral no homogéneo

El criterio de Chong se basa en mirar los autovalores del gradiente de velocidades en el que se busca un par de autovalores complejos, lo que nos lleva a rotación. Podemos expresar la superficie P-Q-R que nos separa las soluciones reales de las complejas por la siguiente ecuación:

$$27R^2 + (4P^3 - 18PQ)R + (4Q^3 - P^2Q^2) = 0 \tag{2.32}$$

Donde si consideramos flujo incompresible, $P = 0$, podemos escribir el plano en función de los invariantes Q y R.

$$D = \frac{27}{4}R^2 + Q^3 \tag{2.33}$$

Donde la curva $D = 0$ separa regiones con autovalores reales con aquellas con pares de autovalores complejos. La condición para que exista un vórtice es que:

$$D > D_{thresh} \tag{2.34}$$

Donde el valor del umbral normalmente se ponía a 0 o a un valor constante. Tal y como se explica en [16], establecer un umbral constante genera problemas de inhomogeneidades en la dirección normal a la pared debido a la presencia de la misma. Cuando se elige un valor adecuado para identificar los torbellinos cerca de la pared, lejos de la misma no se identifica casi ninguno, mientras que disminuimos el umbral para detectar correctamente los vórtices lejanos a la pared, cerca de la misma aparece gran cantidad de puntos que superan el umbral.

Para solucionar este problema, del Álamo [16] propone que el umbral en cada plano dependa la desviación estándar del discriminante, lo que nos lleva a la siguiente ecuación,

$$D(\boldsymbol{x}) > \alpha \overline{D'^2(y)}^{1/2}, \tag{2.35}$$

donde α es el parámetro efectivo de identificación. Con este nuevo umbral la función de densidad de probabilidad de que un punto supere el umbral es homogéneo en todo el canal. Este umbral es representado por del Álamo [16] en la figura 2.1 para Reynolds de fricción entre 180 y 1900.

El valor del parámetro α se elige de tal forma que obtengamos un campo con un tamaño de vórtices idóneo. Si este parámetro se toma muy bajo, muchos puntos del dominio cumplirán el criterio y se unirán entre si formando pocos clúster de gran tamaño, mientras que si se toma un valor muy alto del parámetro solo los puntos que cumplan el criterio de forma muy importante se identificarán, generando estructuras muy pequeñas. El valor que toma del Álamo es $\alpha = 0.02$, el cuál se usará también en nuestras simulaciones.

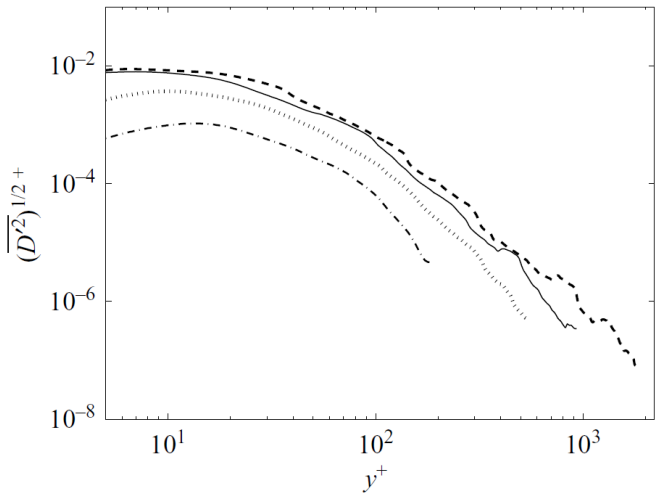


Figura 2.1: Desviación estándar del discriminante en función de la distancia a la pared en mases para diferentes valores de Re_τ. $(- \cdot -)$, $Re_\tau = 180$; $(\ldots)$, $Re_\tau = 550$; $(-)$, $Re_\tau = 950$; $(--)$, $Re_\tau = 1900$.

2.2.2. Eventos Qs

Este método se basa en el análisis cuadrangular, mediante el cual los puntos del dominio se clasifican en función de las fluctuaciones de velocidad del flujo principal y la velocidad perpendicular a la pared. Según el cuadrante en el que se encuentre recibe el nombre de Qs: los eventos Q1 (interacciones externas) que tienen $u > 0$ y $v > 0$, eventos Q2 (*ejections*) que tienen $u < 0$ y $v > 0$, eventos Q3 (interacciones internas) que tienen $u < 0$ y $v < 0$, y los eventos Q4 (*sweeps*) tienen $u > 0$ y $v < 0$. También nos podemos referir a estos eventos como Qs, y se suelen agrupar a los Q2s y Q4s como Q^-, y a los Q1s y Q3s como Q^+ [23].

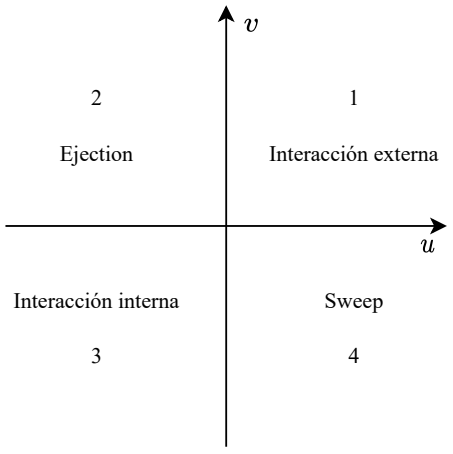


Figura 2.2: Clasificación de los eventos Qs dependiendo del cuadrante

La idea de este método es buscar las estructuras que tienen gran contribución a los esfuerzos de Reynolds. Este primer análisis fue realizado primero en una dimensión en [24], y extendido a las tres dimensiones en [16] y en [23], donde se define como Qs a las regiones conexas que cumplen:

$$|\tau(\boldsymbol{x})| > Hu'(y)v'(y) \tag{2.36}$$

con $\tau(\boldsymbol{x}) = -u(\boldsymbol{x})v(\boldsymbol{x})$ siendo el esfuerzo de Reynolds instantáneo que existe en un punto, y H es el parámetro de este criterio. Los umbrales $u'(y)$ y $v'(y)$ dependen de la distancia a la pared, y esto es debido a que si se tomara un valor constante tendríamos problemas en identificar estructuras en la capa *buffer* o en la región exterior, pero este problema es menor que en el caso de los vórtices identificados por el criterio de Chong. Es por esto que se ha tomado la desviación estándar local, dando buenos resultados en todas las regiones del canal [23].

El parámetro H se puede obtener mediante un estudio de percolación, por el que se ha visto que los resultados son similares en un rango de $1 \leq H \leq 3$. Al igual que en [23], en este trabajo se tomará el valor de $H = 1.75$ que busca maximizar el número de objetos.

Finalmente, cabe destacar que la clasificación de los Qs realizada en la figura 2.2 solo es válido para la mitad inferior del canal. En la mitad superior cambian las definiciones, ya que alejarte de la pared se corresponde con $v < 0$. Si no se cambia esta definición, en la parte superior estaríamos identificando Q1s y Q3s. Este cambio de definición permite que las estructuras de la parte inferior y superior estén conectadas, lo que evita distorsiones en las estadísticas.

Capítulo 3

Herramientas computacionales

En este apartado se van a describir las herramientas necesarias para poder desarrollar los algoritmos de una forma eficiente. Por un lado, necesitamos un lenguaje de programación que, por un lado sea rápido y nos permita administrar la memoria de una forma eficiente, y por otro lado que nos permita albergar las herramientas de paralelización. Un lenguaje muy común en este tipo de problemas es Fortran 90 y es el que se empleará en estas simulaciones.

Para paralelizar los algoritmos se puede hacer uso de *Message Passing Interface* (MPI) de memoria distribuida. Esto quiere decir que cada procesador solo tiene acceso a su propia memoria, y si se requieren de datos de otros procesadores se tiene que hacer a través de mensajes. El otro tipo de paralelización es *OpenMP* de memoria compartida. En este caso, tenemos una serie de procesadores y una única memoria, por lo que todos los procesadores pueden acceder a cualquier dato en cualquier momento. Existen tareas que son óptimas para cada tipo de paralelización, y se usará cada una dependiendo de la aplicación. En el caso de usar ambos tipos al mismo tiempo estaremos hablando de paralelización híbrida, algo que también se usará en nuestros algoritmos como se verá posteriormente.

Otro aspecto importante es un buen uso de la memoria de disco, para una lectura y escritura rápida y eficiente. Para ello se ha usado el estándar HDF5 (*Hierarchical Data Format*), cuyos archivos son bases de datos con *datasets* de n dimensiones, soportando la escritura en paralelo, lo que supone un avance respecto al resto de estándares.

Finalmente se verá de forma simplificada la estructura de un superordenador, en este caso del MareNostrum del *Barcelona Supercomputing Center*, que es el que se ha empleado para realizar los cálculos de este trabajo.

3.1. Fortran 90

Fortran (abreviatura de *FORmula TRANslation*) es un lenguaje de programación de alto nivel nacido en el año 1957 por la empresa IBM, el cual fue uno de los primeros lenguajes de alto nivel que sustituía a la práctica mas empleada en la época, que era usar lenguaje ensamblador. Fortran es un lenguaje enfocado al calculo numérico y aplicaciones científicas, y es ampliamente usado en aplicaciones como la predicción numérica del tiempo, mecánica de fluidos computacional (CFD), elementos finitos o física computacional. Aunque actualmente han nacido multitud de lenguajes de programación, a día de hoy Fortran sigue siendo el lenguaje más importante para implementar algoritmos paralelos [25]. Entre las características mas importantes de Fortran destacan:

- **Compilado:** Esto significa que debemos escribir el programa completo y pasarlo al compilador, antes de poder ejecutarlo. Este compilador generará un archivo ejecutable que será el que se usa para lanzar nuestro programa. En el lado contrario tenemos los lenguajes interpretados, como Javascript o Python, en los que podemos parar la ejecución y ejecutar línea a línea. Esto hace que los lenguajes compilados sean más tediosos y la búsqueda de errores pueda llegar a ser complicada, pero tienen la ventaja que generan archivos ejecutables muy eficientes. De hecho, Fortran tiene distintas *flags* de compilación que pueden hacer el programa muy optimizado, o ralentizarlo para poder comprobar errores más fácilmente. Entre estas *flags* de compilación destacan (para Intel Fortran): `-CB` (Check Bounds), con la cual el programa comprueba que no nos salimos de los límites de ninguna matriz, aunque suponga una ralentización importante; `-O1` `-O2`, `-O3` y `-Ofast`, son distintos niveles de optimización, por el cual el compilador se encargará de modificar el programa para hacerlo lo más rápido y eficiente posible. Dependiendo del tipo de programa, no es extraño que programas en Fortran lleguen a ir uno o dos órdenes de magnitud más rápidos que el mismo programa en Python.

- **Escritura estática:** En Fortran, necesitamos declarar el nombre y el tipo de todas las variables que se van a usar en el programa, y permanecen con el mismo tipo hasta el final del programa. Además, Fortran usa lo que se denomina como escritura estática fuerte, lo que significa que saltará un error si metemos un argumento con un tipo equivocado. Esto tiene una doble ventaja, por un lado ayuda al compilador a hacer más eficiente el programa, y por otro lado ayuda a la identificación de errores.

- **Multi-paradigma**: Podemos escribir programas en Fortran usando diferentes paradigmas o estilos: imperativo, procesal, orientado a objetos o incluso funcional. Dependiendo de la tarea que queramos realizar usaremos un estilo u otro.

- **Paralelo:** Paralelismo es la capacidad de dividir el problema computacional entre distintos procesadores que pueden comunicarse entre ellos. Esta paralelización puede realizarse en un mismo core de procesador (paralelización basada en nodos), diferentes procesadores con misma memoria RAM (paralelización de memoria compartida), o memoria distribuida entre los procesadores de la red (paralelización de memoria

distribuida). Estos procesadores pueden encontrarse en el mismo PC, pueden estar en algún lugar de la sala (como en los supercomputadores), o incluso en cualquier lugar del mundo.

- **Maduro:** Fortran es un lenguaje de programación que tiene más de 60 años de historia, y ha evolucionado en varias versiones: FORTRAN 66, FORTRAN 77, Fortran 90, Fortran 95, Fortran 2003, Fortran 2008 y Fortran 2018. Además, tiene un gran apoyo de la industria debido a su implementación en compiladores, en empresas como IBN, Intel, NVIDIA y otras [25]. El gran cambio se hizo de FORTRAN 77 a Fortran 90, y desde entonces aunque se han añadido diversas funciones, los cambios para programas como el nuestro no son importantes.

Otro aspecto importante de Fortran es saber como maneja la memoria RAM, ya que esto nos permitirá desarrollar los algoritmos de una forma eficiente, evitando saltos del procesador en la memoria. En nuestro caso lo común es usar tensores tridimensionales, siendo cada elemento un punto del dominio, dándonos información sobre alguna variable. Cuando se compila el programa, los datos de estas matrices se alinean en la memoria RAM, siendo contiguos los datos en una única dimensión (como muestra la ecuación (3.1)). Esto hace que el acceso a datos que están contiguos en la primera dimensión sea muy rápido, mientras que si queremos acceder a elementos que estarían contiguos en la tercera dimensión sería mucho más lento.

$$A = \left[\left[\begin{array}{cc} \mathrm{A}_{1,1,1} & \mathrm{A}_{1,2,1} \\ \mathrm{A}_{2,1,1} & \mathrm{A}_{2,2,1} \end{array} \right] \left[\begin{array}{cc} \mathrm{A}_{1,1,2} & \mathrm{A}_{1,2,2} \\ \mathrm{A}_{2,1,2} & \mathrm{A}_{2,2,2} \end{array} \right] \right] \longrightarrow A_{\mathrm{RAM}} = \left[\begin{array}{c} \mathrm{A}_{1,1,1} \\ \mathrm{A}_{2,1,1} \\ \mathrm{A}_{1,2,1} \\ \mathrm{A}_{2,2,1} \\ \mathrm{A}_{1,1,2} \\ \mathrm{A}_{2,1,2} \\ \mathrm{A}_{1,2,2} \\ \mathrm{A}_{2,2,2} \end{array} \right] \tag{3.1}$$

En este caso simple de una matriz $2 \times 2 \times 2$, los datos relativos en la primera dimensión están contiguos en la memoria RAM, cuando están relativos en la segunda dimensión están separados 2 elementos, y en el caso de datos relativos en la tercera dimensión, ya están separados 4 elementos. Este análisis es similar cuando trabajamos con matrices mucho más grandes. Para que el procesador pueda acceder a los datos, estos tienen que ser cargados en caché. Si accedemos a datos contiguos en RAM, estos puedes ser cargados todos al mismo tiempo en caché, con un acceso muy rápido. Sin embargo, si operamos sobre la tercera dimensión, se tendrán que hacer cargas en caché continuamente, con el incremento de coste computacional que esto conlleva.

3.2. HDF5

HDF5 (*Hierarchical Data Format*) es un modelo de datos, librería y tipo de archivo que se usa para el almacenamiento y manejo de datos desarrollado por el *National Center for Supercomputing Applications*. Lo mas interesante de este estándar es que nos permite guardar cualquier tipo de dato de forma muy rápida y eficiente. Además, es compatible con otros software, como por ejemplo MATLAB, lo que resulta de gran interés en nuestro problema. Como su nombre indica, los datos se ordenan de forma jerárquica a través de grupos y datasets, por lo que el proceso de guardado y lectura de datos es muy simple [26].

Un ejemplo de estructura de un archivo de HDF5 se presenta en la figura 3.1. En este tenemos un archivo HDF5 con dos grupos de datos, `header` y `data`, los cuales nos permiten ordenar los datos de la forma que deseemos. Dentro de cada grupo de datos, tenemos los distintos datasets, los cuales pueden ser desde un único dato de tipo *integer* o *double*, hasta matrices de n-dimensiones, por lo que la versatilidad de este tipo de estándar es muy grande. Si quisiéramos acceder, por ejemplo, al `dataset1`, sería mediante la ruta `/data/dataset1`.

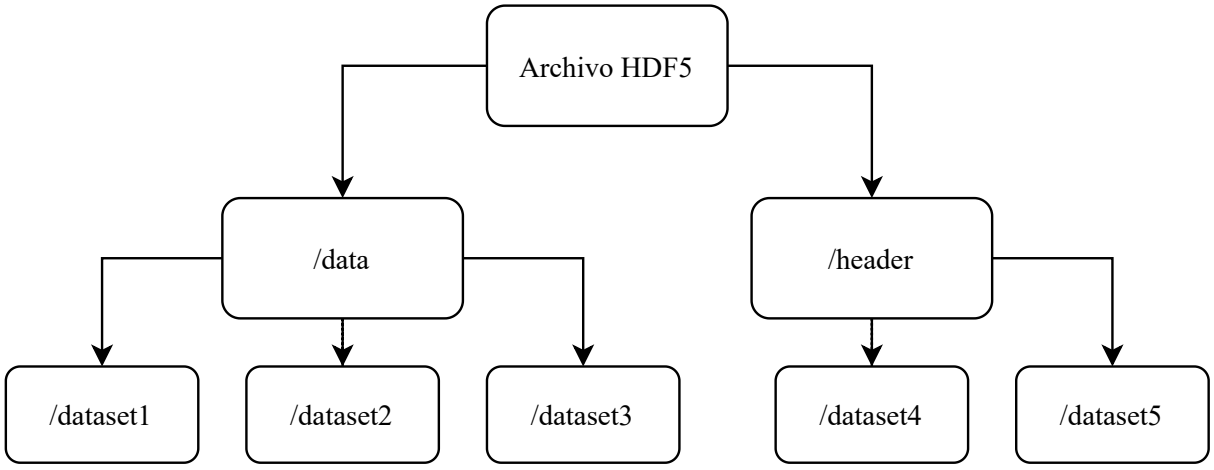


Figura 3.1: Ejemplo de estructura de un archivo típico de HDF5

Se van a usar dos librerías para poder leer y escribir HDF5: *hdf5*, que es la librería general de este estándar, y *h5lt*, HDF5 Lite API, la cual contiene funciones de alto nivel que nos permite hacer más operaciones por llamada que las funciones básicas que vienen en la otra librería. Esto nos permite usar funciones más sencillas que simplifican el proceso de lectura y guardado de datos. A continuación se describirán las distintas funciones que se van a usar en este trabajo.

```
call H5open_f(error)
```

Esta llamada debe realizarse al inicio del programa, y sirve para inicializar la librería HDF5. Esta función (al igual que el resto de funciones de HDF5 en Fortran) tiene un

entero que sirve como comprobación de que la llamada se ha hecho correctamente.

```
call H5fcreate_f(name,flag,file_id,error)
call H5fopen_f  (name,flag,file_id,error)
```

La primera de estas funciones sirve para crear un archivo nuevo, y la segunda para abrir una archivo HDF5 ya existente. En el caso de crear un nuevo archivo, mediante la `flag` podemos indicar que sobrescriba un archivo ya existente, o que salte un error en caso de que el archivo ya exista. En el caso de abrirlo, con esta `flag` indicamos si queremos el archivo de solo lectura, o lectura y escritura. En ambos casos, generamos un identificador que nos permitirá acceder al archivo.

```
call H5gcreate_f(file_id,name,group_id,error)
call H5gopen_f  (file_id,name,group_id,error)
```

Estas funciones son análogas a las anteriores pero en este caso es para crear o abrir grupos de datos. En este caso se introduce el identificador del archivo y nos devuelve el identificador del grupo.

```
call H5LTmake_dataset_int_f   (id,name,rank,size,buffer,error)
call H5LTmake_dataset_double_f(id,name,rank,size,buffer,error)
call H5LTmake_dataset_string_f(id,name,rank,size,buffer,error)
```

Las funciones mostradas sirven para crear datasets, cada una de un tipo distinto de dato. Se pasa el identificador del grupo o archivo deseado, el nombre que queremos poner a la variable en la base de datos, la dimensión, el tamaño de las matrices y finalmente la variable de nuestro programa que queremos guardar. Como usuarios no nos tenemos que preocupar cómo ordenar los datos dentro del archivo, de esto se encarga la propia librería. Aunque se ha mostrado únicamente las funciones para guardar enteros, reales y cadenas de texto, existen las funciones análogas para otros tipos de datos, visitar el manual [27].

```
call H5LTread_dataset_int_f   (id,name,buffer,size,error)
call H5LTread_dataset_double_f(id,name,buffer,size,error)
call H5LTread_dataset_string_f(id,name,buffer,size,error)
```

De forma análoga a las funciones anteriores, estas sirven para leer los datos ya guardados en en un archivo HDF5. En este caso no es necesario especificar el rango, debido a la forma en la que Fortran administra la memoria. Como se ha visto en la sección §3.1, la memoria se almacena de forma vectorial, por lo que cambiar la dimensión de una variable es trivial, no requiere de ninguna operación.

```
call H5gclose_f(id,error)
call H5fclose_f(id,error)
```

Finalmente tenemos las funciones que cierran los grupos y el archivo respectivamente, en las que únicamente tenemos que indicar el identificador de cada elemento.

Otra característica que hace HDF5 el estándar apto para nuestros algoritmos es que permite guardar datos en paralelo, es decir, si tenemos los datos de una variable dividida entre distintos procesadores, cada procesador puede guardar su parte. Esto es una característica muy remarcable cuando estamos trabajando con paralelización de memoria distribuida, porque acelera la escritura de datos por dos motivos: en primer lugar están todos los procesadores escribiendo al tiempo una misma variable, y por otro lado evitamos las comunicaciones a un solo nodo que se tiene que encargar de unificar la variable y escribirla ella misma. Cuando se trabaja con variables muy grandes, y además en un supercomputador, esto se acentúa, ya que si los procesadores se encuentran muy alejados entre sí la comunicación puede ralentizarse (esto se explicará en mas detalle en la sección §3.5). Crear las funciones que permiten la escritura en paralelo no es trivial y se han tomado las funciones ya existentes en el departamento, por lo que no entraremos en profundidad.

3.3. MPI

El estándar del *Message Passing Interface* (MPI) es ampliamente usado para aplicaciones de paralelización mediante memoria distribuida. Este estándar se puede implementar en programas en C, C++ y Fortran. El objetivo de MPI es crear algoritmos muy eficientes, escalables y portables, y es por esto que sea el modelo más usado en *high-performance computing* actualmente. El hecho de que sea memoria distribuida implica que cada procesador tiene su propia unidad de memoria independiente del resto. Si un procesador necesita de información que contiene otro procesador, estos datos se tienen que enviar mediante una comunicación. El proceso que se sigue a la hora de plantear un algoritmo mediante MPI es el que se muestra en la figura 3.2.

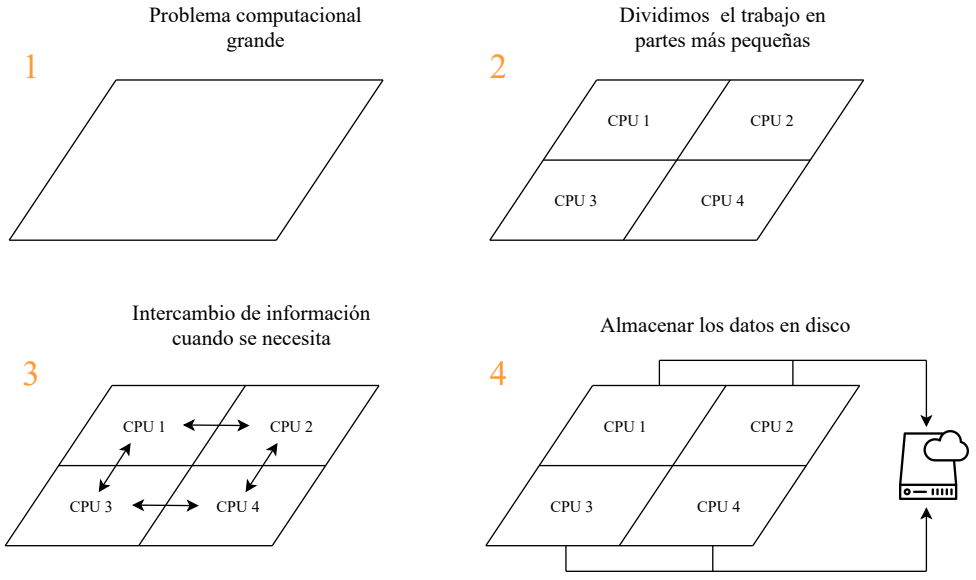


Figura 3.2: Esquema de un algoritmo genérico de MPI

Inicialmente tenemos un problema suficientemente grande que necesitaría mucho tiempo para ser resuelto con un único procesador, se divide este problema en partes más pequeñas que se van a repartir entre los distintos procesadores, procurando que la carga de trabajo sea equitativa entre ellos. En el momento que se necesite algún tipo de dato de otro procesador se establecen comunicaciones entre ellos. Finalmente, cuando el problema esta resuelto se guardan los resultados en disco para su posterior post-procesado.

Este tipo de paralelización es especialmente idónea para los cálculos de mecánica de fluidos debido a la forma en la que se resuelven computacionalmente las ecuaciones de Navier-Stokes. Cuando mallamos el dominio y resolvemos las ecuaciones para cada celda, se necesitan los datos de la celda actual y de las adyacentes. Al paralelizarlo mediante MPI el dominio computacional se divide entre todos los procesadores. Las celdas interiores podrán resolverse sin necesidad de realizar ninguna comunicación, pero en las celdas frontera es necesaria la comunicación entre procesadores. Es por esto que cuanto menor sea el área de la frontera entre dos procesadores, menos comunicaciones se requieren y por lo tanto aumenta la velocidad de cálculo.

Otro aspecto fundamental que tienen que tener nuestros algoritmos es que el reparto de trabajo tiene que ser lo más equitativo posible independientemente del número de procesadores que pongamos al cálculo. Lo mas normal es que el tiempo que tarde nuestro programa en ejecutarse sea igual a la velocidad del procesador más lento. Si sobrecargamos a un procesador, no importa que el resto de procesadores terminen muy rápido, ya que hasta que no termine este último no habrá terminado nuestro programa.

Cuando se haga la implementación de los algoritmos de identificación y seguimiento de vórtices junto al código LISO se crearán distintos grupos de procesadores, cada uno dedicado exclusivamente a una parte del código. Cuando se inicia el programa, se inicializa MPI a través de un comunicador en el que están todos los procesadores, llamado `MPI_COMM_WORLD`. A través de este comunicador global se pueden comunicar todos los procesadores con todos, y es el que normalmente se usa en la mayoría de aplicaciones.

Cuando se requieren del uso de grupos de procesadores, se hacen particiones de este comunicador global en otros mas pequeños, teniendo un comunicador personal para cada grupo. Además, en el caso de que existan comunicaciones de datos entre distintos grupos, se crean intercomunicadores, que nos permite comunicar de manera sencilla los procesadores de uno y otro grupo. Todo esto se representa de manera esquemática en la figura 3.3. En esta figura tenemos en total 4 comunicadores: el comunicador global, el comunicador propio de cada grupo `comm` que tiene el mismo nombre en todos los grupos pero solo relaciona procesadores del mismo grupo, y dos comunicadores que comunican grupos, `interg12` y `interg23`.

Esta arquitectura es la que se va a usar en nuestro algoritmo, donde el grupo 2 es una memoria virtual que se encarga de almacenar datos e ir pasando información de un grupo a otro, y los grupos 1 y 3 son los que se encargan de resolver la DNS y la identificación de vórtices respectivamente. Es posible también hacer la intercomunicación entre grupos usando el comunicador global, pero al hacer los grupos la identificación de procesadores

cambia a un nuevo identificador local, por lo que hacer la comunicación global es más complicado.

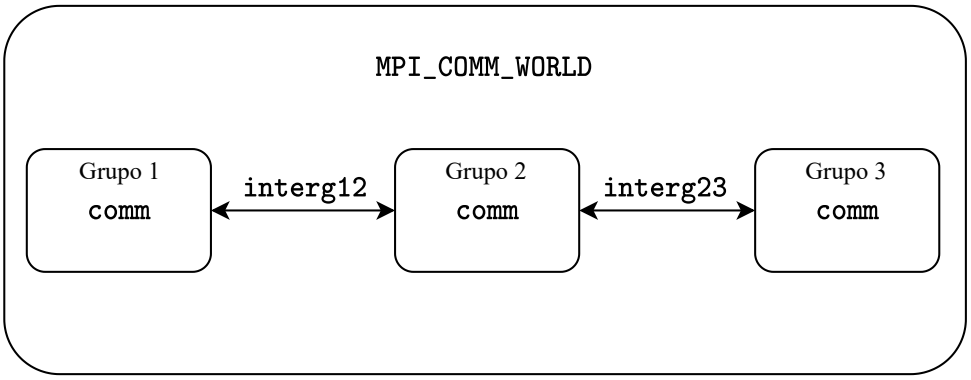


Figura 3.3: Comunicadores en un código con grupos MPI

Esta arquitectura es básica pero funciona para el tipo de paralelización que nosotros necesitamos. Se podrían hacer grupos dentro de otros grupos y hacer estructuras más complejas, pero en nuestro caso no es necesario. A continuación detallamos las órdenes de MPI que usaremos a lo largo de este proyecto.

```
call MPI_INIT(ierr)
```

Esta llamada se tiene que realizar al inicio del programa cuando se quiera usar MPI. Al igual que ocurría con HDF5, estas funciones también tienen un entero que nos indica si ha sucedido algún error durante la llamada.

```
call MPI_COMM_RANK(comm,myid,ierr)
call MPI_COMM_SIZE(comm,nummpi,ierr)
```

Estas funciones sirven para asignar a cada procesador su propio ID que conservarán a lo largo del programa, y saber cuantos procesadores son en ese comunicador, a través de la variable `nummpi`. Estas llamadas se realizan al inicio del programa con el comunicador `MPI_COMM_WORLD`, pero una vez divida este en varios grupos, se debe de llamar otra vez a dichas funciones con el nuevo comunicador, que asignarán un identificador nuevo a cada procesador y notificará a todos los procesadores cuantos son en dicho grupo. Por ejemplo, si tenemos un cálculo con 4 procesadores, estos van a recibir un ID global entre el 0 y el 3; si ahora dividimos estos 4 procesadores en dos grupos de 2, cada grupo tiene su propio ID, por lo que ahora tendremos dos procesadores con ID = 0 y ID = 1 pero con un comunicador distinto, evitando así la confusión.

```
call MPI_COMM_SPLIT(comm,color,id,new_comm,ierr)
```

Esta es la llamada que nos permite dividir el comunicador `comm` en varios grupos. Antes, tenemos que asignar a cada procesador el valor de `color` = 1,2,3... en función el número de grupos que queramos crear, siendo los procesadores con el mismo valor de color lo que

van a pertenecer al nuevo grupo. Esto genera el nuevo comunicador, con el cual debemos volver a llamar a las funciones `MPI_COMM_RANK` y `MPI_COMM_SIZE` para asignar los nuevos IDs de cada grupo y obtener el número total de procesadores que forman el grupo (aunque esta llamada no es totalmente necesaria, ya que el número de procesadores por grupo lo hemos establecido nosotros a través de `color`).

```
call MPI_INTERCOMM_CREATE(local_comm,local_master,peer_comm,remote_master,&
& tag,new_intercomm,ierr)
```

Esta es la función encargada de crear los comunicadores entre grupos. Por orden, necesitamos el comunicador local del grupo, el ID local del procesador que va a ser el *master* (normalmente se corresponde con el procesador con ID=0), el comunicador global que tienen en común ambos grupos (en el caso de la figura 3.3, sería `MPI_COMM_WORLD`)), el ID global del procesador *master* del otro grupo, un *tag* para identificar distintas creaciones de intercomunicadores, y finalmente el intercomunicador que deseamos. Todos los procesadores de cada grupo tiene que entrar en su correspondiente llamada a la función. Si tenemos un caso con 4 procesadores, ID=0,1 pertenecientes al grupo 1 y ID=2,3 pertenecientes al grupo 2, la llamada que haga el grupo 1 a la función tendrá que poner como `remote_master` el procesador con ID=2, que tendrá ID=0 en locales, mientras que el grupo 2 pondrá ID=0 en esta casilla.

Hasta aquí hemos visto las funciones que configuran nuestro caso MPI, ahora procedemos a ver distintas formas de comunicación entre procesadores.

```
call MPI_SEND(buffer,count,datatype,dest  ,tag,comm,ierr)
call MPI_RECV(buffer,count,datatype,source,tag,comm,status,ierr)
```

Estas son las dos funciones básicas de MPI para poder enviar y recibir datos de forma síncrona (esto quiere decir que tanto el procesador emisor como el receptor deben estar dentro de la función). Como podemos ver, la comunicación es muy simple: en primer lugar necesitamos la variable que vamos a enviar, después el número de elementos que se van a enviar sin dimensiones, debido a como Fortran guarda los datos en memoria; luego introducimos el tipo de dato que estamos mandando y dependiendo si estamos enviando o recibiendo, el destino o el origen de la comunicación. Finalmente ponemos un *tag* que nos ayuda a diferenciar distintos envíos y el comunicador a partir del cual se va a hacer la comunicación. Existe la función `MPI_SENDRECV` que sirve para enviar y recibir datos entre dos procesadores, muy útil cuando el intercambio de datos es bidireccional.

```
call MPI_BCAST(buffer,count,datatype,root,comm,ierr)
```

Esta función realiza una difusión de una variable al resto de procesadores del grupo. Normalmente se utiliza cuando mandamos a un procesador a hacer una lectura de datos de un fichero y tiene que transmitirlos al resto de procesadores. Esta función realiza los envíos de manera muy rápida, ya que sigue una estructura de árbol: un procesador manda los datos a otro, y a su vez esos dos los mandan a otros dos y continua.

```
call MPI_REDUCE   (sendbuff,recvbuff,count,datatype,op,root,comm,ierr)
call MPI_ALLREDUCE(sendbuff,recvbuff,count,datatype,op,comm,ierr)
```

Las funciones `MPI_REDUCE` y `MPI_ALLREDUCE` se utilizan para realizar una operación sobre la variable que deseemos. El tipo de operaciones que se pueden realizar son múltiples, pero destacan la suma, el producto, el máximo, mínimo y todo tipo de operaciones booleanas. En la figura 3.4 [28] se muestra un ejemplo de como usar esta función usando la suma. La diferencia entre ambas funciones es que en la función `MPI_ALLREDUCE` el resultado de la operación lo guardan todos los procesadores, y en la función `MPI_REDUCE` el resultado solo se almacena en el procesador especificado en `root`.

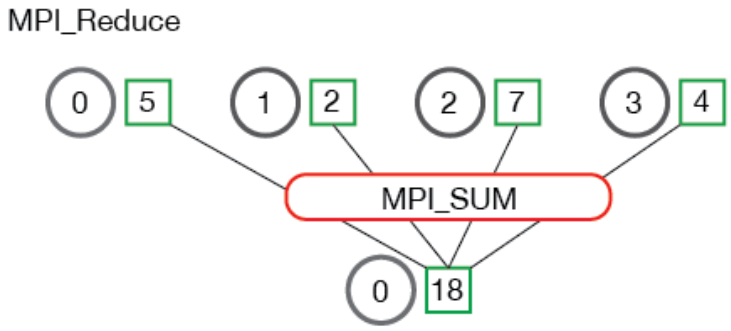


Figura 3.4: Ejemplo del uso del `MPI_REDUCE` con la suma como operación

```
call MPI_BARRIER(comm,ierr)
```

Como su nombre indica, esta función actúa como barrera. Hasta que no llegan todos los procesadores del comunicador hasta la línea en la que se encuentra esta función, no avanza ninguno. Esto se utiliza cuando se necesita haber terminado una tarea por completo antes de comenzar la siguiente.

```
call MPI_FINALIZE(ierr)
```

Finalmente, esta función termina el uso de MPI. Además de las funciones que se han explicado, existen gran variedad de funciones que pueden ayudarnos para cualquier operación que queramos realizar. Toda la documentación relativa a MPI la podemos encontrar en [29].

3.4. OpenMP

OpenMP es un estándar que nos permite paralelizar nuestros algoritmos mediante memoria compartida. Al igual que con MPI, se puede implementar en lenguajes como C, C++ y Fortran. La paralelización por memoria compartida permite que tengamos pro-

cesadores independientes compartiendo una misma unidad de memoria. Cada procesador puede acceder a cualquier localización de memoria en todo momento, y cada procesador puede ejecutar diferentes instrucciones en distintos datos debido a que cada procesador tiene su propia unidad de control. Las CPUs actuales cuentan con procesadores multi-core, esto quiere decir que tienen unidades de procesamiento diferentes llamadas *cores*. Además, cada *core* tiene la capacidad de ejecutar varios hilos de ejecución al tiempo, también llamados *threads* [30].

Al igual que todos los tipos de paralelización, esta tiene sus ventajas y sus inconvenientes. Como todos los procesadores tienen acceso a toda la memoria, no necesitan usar mensajes para transferir información, por lo que este tipo de paralelización es muy rápida. Sin embargo, podemos tener problemas de sincronización si dos procesadores intentan escribir sobre una misma variable al mismo tiempo.

Imaginemos un caso con dos hilos de ejecución, el primero va a incrementar el valor de una variable por 1 y el otro va a incrementar la misma por 2, por lo que al final el valor de la variable debe incrementarse por 3. El proceso que siguen los procesadores es el siguiente: cada procesador lee el valor que tiene la variable, aplica la operación que indique la línea de programación, y después escribe el resultado en la misma variable. Si esta operación se realiza sin ningún control sucede lo que se muestra en la figura 3.5, ambos hilos leen el mismo valor de memoria, realizan la operación y guardan su resultado, y este será un incremento de 1 o 2 dependiendo de cuál hace la ultima escritura, pero en ningún caso obtenemos el resultado de 3 como deseábamos. Una buena paralelización es la mostrada en la figura 3.6, donde se realiza un bloqueo de la variable en la que se va a operar. En este último caso, hasta que un procesador no ha acabado su proceso de lectura y escritura, el siguiente se queda esperando.

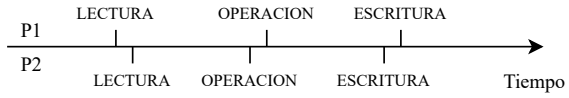


Figura 3.5: Mala implementación de la paralelización OpenMP

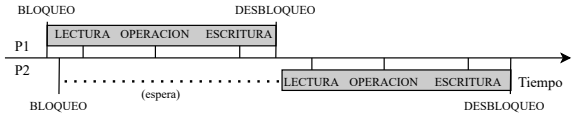


Figura 3.6: Implementación correcta de la paralelización OpenMP

Como hemos podido ver, la forma de paralelizar es totalmente distinta a la que se realiza en MPI. OpenMP es idóneo cuando hay que realizar operaciones sobre un vector en el que cada procesador accede a una única posición de memoria, sin que el resto de procesadores necesiten leerla. Si tenemos muchas variables compartidas por todos los procesadores, tendremos que hacer uso de esperas entre procesadores, lo que ralentiza nuestro algoritmo. Es posible usar ambas paralelizaciones al mismo tiempo, teniendo distintos procesadores corriendo por nuestro algoritmo y en ciertas situaciones abrimos regiones paralelas, donde cada procesador abrirá sus propios hilos, independientes del resto de procesadores, compartiendo únicamente la memoria de su propio procesador.

En Fortran todas las directivas de OpenMP comienzan por `!$OMP`. Como el primer carácter es "!" y el compilador normalmente lo interpreta como comentario, si al compilarlo no

indicamos que vamos a ejecutar con OpenMP estas instrucciones serán ignoradas. A continuación mostramos las órdenes que se usarán en nuestros algoritmos.

```
!$OMP PARALLEL
!$OMP END PARALLEL
```

Esta es la directiva más importante de OpenMP y es la que se encarga de abrir regiones paralelas. Estas regiones son bloques de código que van a ser ejecutadas por múltiples hilos en paralelo. El código que va antes y después de la región paralela se ejecuta por un único hilo, y recibe el nombre de región en serie. Además, se pueden especificar una serie de condiciones al abrir la región paralela. La más importante y la que se usará en este trabajo es `PRIVATE`(*list*). Todas las variables que se especifiquen dentro de este paréntesis serán privadas para cada procesador a lo largo de la región paralela, es decir, se generan copias locales para cada hilo de ejecución. Para conocer más información se puede visitar [31].

Con la región paralela, lo único que hemos conseguido es que todos los hilos que se han abierto ejecuten exactamente la misma tarea, lo cual no es un objetivo de la paralelización. Es por esto que tenemos que ver distintos constructores que nos permiten distribuir la carga de trabajo entre los distintos hilos. El constructor más importante que existe en OpenMP es el siguiente:

```
!$OMP DO
do i = 1,1000
...
enddo
!$OMP END DO
```

Este constructor se encarga de dividir el bucle do entre los distintos procesadores para poder ejecutarlo en paralelo, y se representa de forma esquemática en la figura 3.7. Como podemos ver, el reparto de la carga de trabajo no lo realiza el usuario, sino que es el propio estándar el que se encarga de hacer este reparto, lo que facilita la tarea al programador. Al igual que con la región paralela, este bucle puede tener distintas condiciones, sobre todo para especificar alguna variable adicional como privada.

En la figura 3.7 tenemos el bucle do anterior representado de forma esquemática, suponiendo que tras la paralelización tenemos 10 hilos. Este bucle se reparte en partes iguales entre los distintos hilos de ejecución, aumentando la velocidad del algoritmo. En este caso, la variable `i` deberá ser declarada como privada, ya que esta tiene valores distintos dependiendo del hilo en el que nos encontremos.

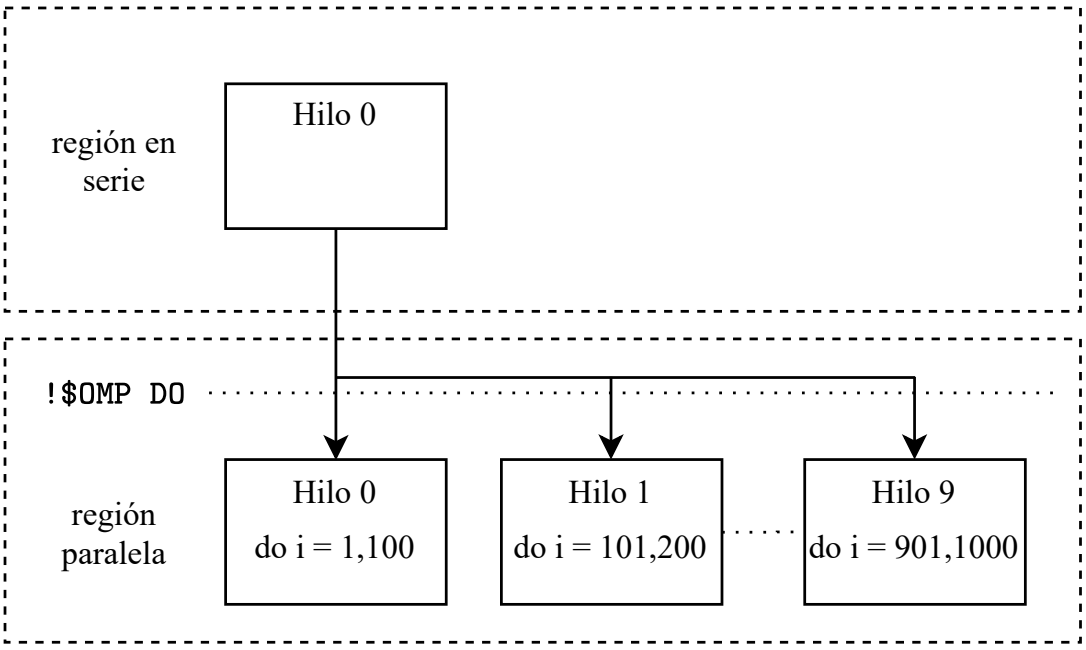


Figura 3.7: División de la carga de trabajo usando `!$OMP DO`

Otro constructor muy importantes es el `!$OMP SINGLE`.

```
!$OMP SINGLE
!$OMP END SINGLE
```

El código que está dentro de esta directiva es ejecutado únicamente por un único hilo, y entra el que primero llega. El resto de procesadores esperan a que este procesador salga de este constructor ya que lleva implícito una sincronización. Este constructor es usado normalmente para inicializar variables compartidas, reservar memoria o liberar memoria, pero siempre en variables públicas.

Finalmente, nos quedan ver un par de directivas que nos permiten sincronizar el código. Comenzamos por `!$OMP CRITICAL`.

```
!$OMP CRITICAL
!$OMP END CRITICAL
```

El código que se encuentra dentro de esta directiva restringe el acceso a un procesador al mismo tiempo. Así nos aseguramos que lo que se está haciendo se hace correctamente. El ejemplo más claro es el mostrado en las figuras 3.5 y 3.6, y es cuando se requiere actualizar una variable pública. También es muy usual cuando se requiere algún input por teclado o archivo o escribir datos a disco. Un atributo importante que debemos asignar a las secciones críticas es un nombre, ya que si existen diferentes secciones críticas con el mismo nombre, un único hilo va a estar dentro de ellas al mismo tiempo. Incluso si no especificamos ningún nombre, todas aquellas que no tengan nombre se tratarán con una única sección crítica.

```
!$OMP BARRIER
```

Finalmente tenemos la directiva `!$OMP BARRIER`, cuya función es análoga a la ya vista en MPI llamada `MPI_BARRIER` y sirve para sincronizar hilos diferentes en el equipo. Al llegar a esta línea, todos los hilos esperan a que el resto hayan llegado a este punto.

3.5. Supercomputación

Un superordenador es un ordenador de alto rendimiento, compuesto por un gran número de procesadores y una red de alta velocidad. La forma de medir el rendimiento de estos superordenadores es a través del número de operaciones de coma flotante que pueden hacer por segundo (*FLOPS*). A día de redacción de este trabajo, el mayor supercomputador del mundo es el *Supercomputer Fugaku*, localizado en Japón, con una potencia de 415530 TFlops/s, con casi 7.3 millones de cores [32]. En nuestro caso nos centraremos en la estructura del superordenador localizado en Barcelona, llamado MareNostrum, del *Barcelona Supercomputing Center*, siendo el más potente de España, y el 37 más potente a nivel global. Este será el superordenador que se usará para realizar los cálculos de este proyecto.

Desde el nacimiento de MareNostrum en 2005, siendo el más potente de Europa y uno de los mejores en todo el mundo, ha sido actualizado en varias versiones. Actualmente nos encontramos en MareNostrum 4, que entró en operación en 2017, y a finales de este año 2020 se incorporará la nueva versión MareNostrum 5. El ordenador actual cuenta con procesadores Intel Xeon Platinum de la generación Skylake, organzados en racks de cómputo Lenovo SD530, basados en el sistema operativo Linux y una red de alta velocidad Intel Omni-Path. Actualmente tiene una potencia de 6.2 Petaflops.

El bloque general de este supercomputador contiene 48 racks de cómputo, albergando un total de 3456 nodos de cálculo, dando un total de 165888 procesadores con una memoria principal total de 390 TB. Las características son las siguientes [33]:

- 2 Intel Xenon Platinum por nodo, con 24 cores cada uno, lo que da 48 cores en total por nodo.
- 3240 nodos con 12x8 GB DDR4-2667 DIMMS, lo que da a unos 2GB por core y 96GB por nodo.
- 216 nodos de alta memoria, con 12x32 GB DDR4-2667 DIMMS, lo que da unos 8GB por core y 384GB por nodo.
- 100 Gbit/s en la red de comunicaciones Intel Omni-Path.
- 10 Gbit/s en la red Ethernet.

La velocidad de nuestros algoritmos depende de la velocidad de cálculo más la velocidad en las comunicaciones. Los procesadores que se encuentran en el mismo nodo establecen la comunicación muy rápidamente, pero conforme se alejan entre sí, establecer la comunicación cada vez es más lento. Si queremos maximizar la eficiencia de nuestros algoritmos, hay que reducir el número de comunicaciones que se realizan y aumentar el número de datos que se transmiten a través de ella (ya que la velocidad de transmisión de datos es muy rápida).

Como hemos podido ver en las características, la memoria RAM por nodo es muy alta, teniendo incluso nodos de alta memoria, con más de 380GB por nodo. Esto se va a aprovechar para realizar una memoria virtual dentro de nuestros algoritmos, evitando escribir a disco, aumentando la velocidad general de nuestros algoritmos. Esto se verá mas en profundidad en el capítulo de integración con el programa LISO.

Capítulo 4

Identificación de estructuras coherentes

Como ya se ha explicado anteriormente, una estructura coherente o vórtice es una serie de puntos del dominio que cumplen un criterio de vorticidad establecido y además se encuentran conectados entre sí siguiendo las 3 direcciones cartesianas. Para que sea una estructura coherente, debe ser posible conectar dos puntos cualquiera de la estructura pasando únicamente por otros puntos de la estructura y siguiendo las direcciones cartesianas. También se puede definir que un punto pertenece a una estructura coherente si al menos contiene un punto vecino de la estructura coherente en cualquier dirección recta cartesiana.

Empezaremos el capitulo con la descripción del problema y el concepto general del algoritmo, para posteriormente entrar en profundidad en las particularidades del mismo.

4.1. Descripción del problema

Para este programa vamos a tener como entradas la matriz tridimensional del tamaño del dominio, formada por 0 y 1 en función de si dicho punto supera el criterio de vorticidad o no. Además, también tenemos como entrada las características geométricas del dominio. Este algoritmo se ampliará de forma muy sencilla para distinguir distintos elementos, como por ejemplo *sweeps* y *ejections*, manteniendo la esencia principal del algoritmo pero aportando mucha más información.

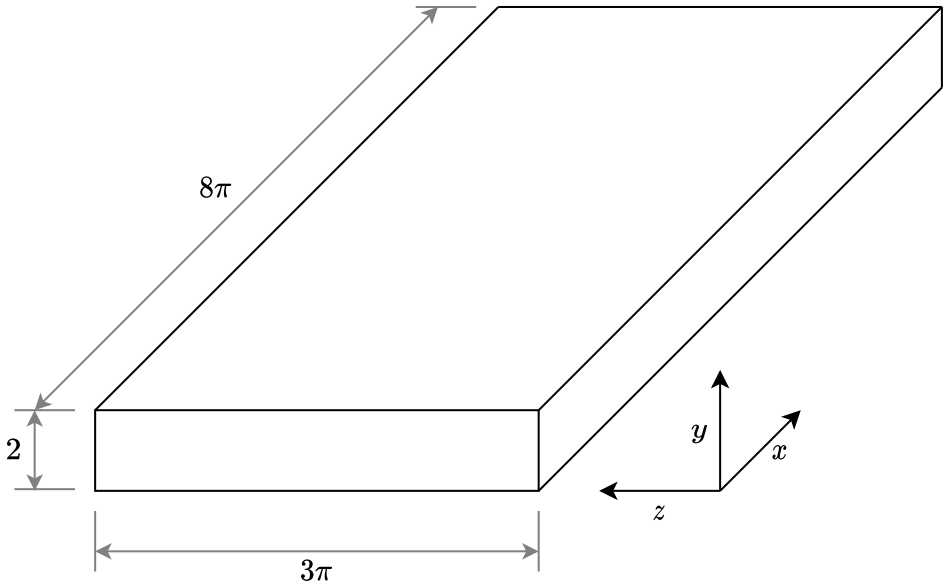


Figura 4.1: Geometría del canal a estudiar

En este caso se harán simulaciones a dos Reynolds, $Re_\tau = 500$ y $Re_\tau = 1000$, teniendo en ambos casos la geometría mostrada en la figura 4.1. El flujo movido es movido por una gradiente de presiones, lo que indica que estamos ante un flujo de Poiseuille. Tenemos una altura del canal de $2h$, mientras que en la dimensión longitudinal tenemos una longitud de 8π y en la transversal de 3π, elegidos como múltiplos de π para mantener la periodicidad en Fourier. El resto de características se muestran en la tabla 4.1.

Características del fluido			
Número de Reynolds de fricción	Re_τ	500	1000
Número de Reynolds	Re	11480	20580

Características Geométricas			
Puntos en dirección x	N_x	1536	3072
Puntos en dirección y	N_y	251	383
Puntos en dirección z	N_z	1152	2304
Longitud en dirección x	L_x	8π	8π
Longitud en dirección z	L_z	3π	3π
Altura del canal	h	2	2

Características de la Simulación			
Condición de Courant-Friedrichs-Levy	CFL	0.9	0.9
Paso temporal	Δt	0.0295	0.0146

Tabla 4.1: Características de las simulaciones empleadas

El espaciado de la malla es diferente dependiendo de la dirección. En dirección x y z

el espaciado es uniforme y se puede calcular de forma inmediata como $\Delta x = L_x/N_x$ y $\Delta z = L_z/N_z$. Debido a la capa límite, el mallado no es uniforme en dirección y, por lo que tenemos un vector con el mallado en esta dirección. De esta forma, el espaciado se puede calcular de la siguiente forma,

$$\Delta y^{(i)} = \frac{y^{(i+1)} - y^{(i-1)}}{2}, \tag{4.1}$$

siendo estos espaciados muy importantes para calcular el volumen y centros de masas de los vórtices cuando saquemos sus características geométricas. Con todos los datos mostrados, ya tenemos el punto de partida para nuestro algoritmo.

Para la identificación de vórtices se partirá de un punto semilla o germen, a partir del cual se mirarán en sus 6 direcciones cartesianas (ver figura 4.2) para ver si existe otro punto conexo. Una vez hemos identificado todas sus conexiones, saltamos al siguiente y volvemos a hacer lo mismo de forma recurrente, hasta que se ha identificado el vórtice por completo. Como casos especiales tenemos las celdas que se encuentran sobre una cara de simetría, que tendrían que comprobar si existe punto vórtice en el otro lado del dominio, y las celdas que se encuentran sobre la pared, las cuales tienen una dirección menos donde comprobar la conectividad.

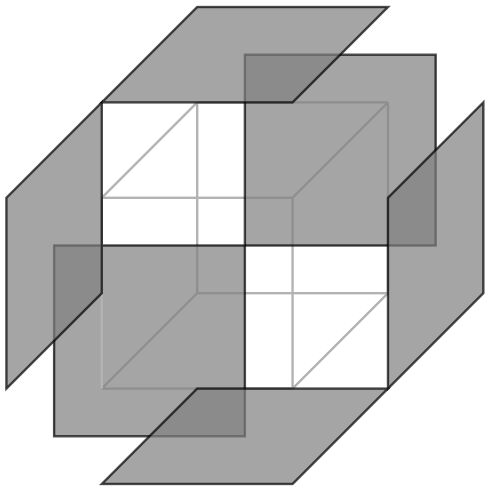


Figura 4.2: Conectividad de una celda con el resto del dominio

Este primer algoritmo estará paralelizado mediante MPI, y para ello lo primero que se va a realizar es la división del dominio entre el número de procesadores partiéndolo en la dirección x, teniendo planos YZ completos. En estos problemas la malla en x es una potencia pura de 2 y el número de procesadores que utilizaremos también lo será, por lo que la división equitativa del trabajo es inmediata. Cada procesador hará la identificación de vórtices en su propio subdominio, pero puede ocurrir que al realizar la división del dominio partamos algunos vórtices, conteniendo partes de los mismos en varios subdominios, como podemos ver en la figura 4.3. Vamos a tener partes del mismo vórtice en varios

procesadores, por lo que se realiza una unificación de todos los vortices, teniendo la lista definitiva de los mismos.

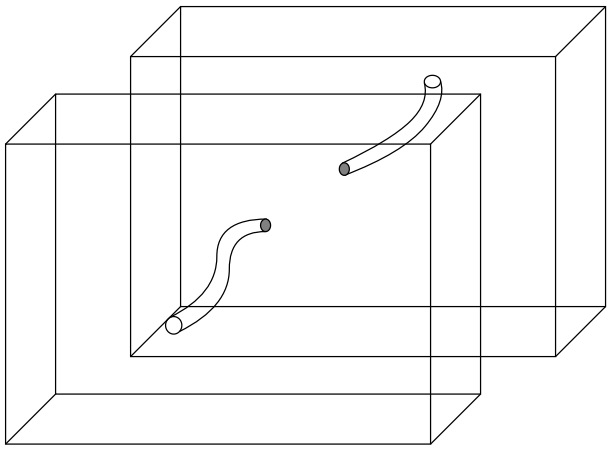


Figura 4.3: Ejemplo de vórtice separado por la división del dominio

Finalmente, se obtienen todas las características de los mismos, ya pueda ser el volumen, el centro de masas, la caja que envuelve al vórtice, etc. Hasta aquí hemos visto la parte conceptual del algoritmo desarrollado, y en la siguiente sección se verá de manera más profunda la organización de los datos para maximizar la eficiencia y las distintas comunicaciones entre los procesadores, además de ver de forma más detallada como se ha desarrollado el algoritmo.

4.2. Algoritmo de identificación 3D

Como se ha visto en el apartado anterior, este algoritmo consta de las siguientes partes, en orden:

- Lectura de datos por parte del procesador maestro y transmisión de la información al resto de procesadores esclavos.
- Identificación de vórtices por cada procesador en sus respectivos subdominios.
- Unificación de todos los vórtices, uniendo aquellos que fueron separados al dividir el dominio.
- Filtrado de los vórtices y obtención de sus características: volumen, centro de masas, tamaño de la caja envolvente al vórtice, y su posición en el dominio.
- Guardado de datos a disco.

Una vez se ha inicializado MPI y se ha hecho la lectura de datos por parte del procesador principal, este envía a cada procesador la parte del dominio que le corresponde únicamente. Para ello tenemos un total de planos YZ igual a N_x y un número de procesadores igual a `nummpi`, por lo que:

$$\texttt{div} = \frac{N_x}{\texttt{nummpi}} \tag{4.2}$$

El procesador maestro se quedará con los planos [1,`div`], el segundo tendrá los planos [`div`+1, 2·`div`] y así sucesivamente hasta que cada procesador tiene únicamente la parte del dominio que le corresponde. En este punto el procesador maestro puede liberar de memoria la matriz total del dominio.

4.2.1. Agrupación de estructuras

Este algoritmo va a tener por salida una matriz en la que se guardan las coordenadas (x,z,y) en unidades de malla de los puntos que conforman cada vórtice. No existe una matriz por vórtice, sino que existe una matriz única, y para poder distinguir un vórtice de otro se deja una linea en blanco. Para mejorar la eficiencia, cada coordenada se escribe en una fila, y cada punto se van escribiendo en distintas columnas, lo que hace que las tres coordenadas de cada punto estén seguidas en memoria.

El bucle principal de agrupación de estructuras es el siguiente:

```
do kk = 1,ny
    do jj = 1,nz
        do ii = 1,div
            if (B(ii,jj,kk) .eq. 1) then
                nvor = nvor + 1
                p = (/ii,jj,kk/)
                call agregado(p,myid)
                pos = pos + 1;
            endif
        enddo
    enddo
enddo
```

Por la forma en la que nos vienen los datos del código DNS, el primer índice es la coordenada x, el segundo es la coordenada z y el último es la coordenada y, por lo tanto, al realizar el triple bucle do, el bucle interno es el que itera sobre el primer índice, para ir mirando el posiciones consecutivas de memoria, ahorrando así tiempo de lectura de datos. La matriz `B` es el subdominio, y cuando encontramos un 1 ya tenemos el punto semilla para encontrar todo el vórtice, por lo que aumentamos el número de vórtices, guardamos el punto que hemos encontrado, y llamamos a la subrutina de agregado, que meterá todo

el vórtice en nuestra matriz de coordenadas. El hecho de tener que incluir un condicional dentro de un triple bucle do penaliza la eficiencia, por lo que el apartado de agregado toma tiempo.

La subrutina de agregado lo que hace es recibir un punto y lo mete a una cola. La cola es una matriz donde vamos almacenando los puntos antes de que se procesen y se añadan a la matriz final de vórtices. Como en Fortran hay que especificar el tamaño de las variables antes de usarlas, y la cola no va a ser del mismo tamaño para todos los vórtices, se define una variable con memoria suficiente para el caso del vórtice más grande que se pueda obtener. Además de la variable de la cola necesitamos dos índices, uno que vaya indicando cuantos elementos tenemos en la cola, y otro que nos indique que punto estamos analizando. En el momento en el que el índice del punto que toca analizar sea mayor que el índice de máximos elementos en la cola, habremos identificado por completo el vórtice y podremos salir de la función. El diagrama de flujo que representa el funcionamiento de la función es el siguiente:

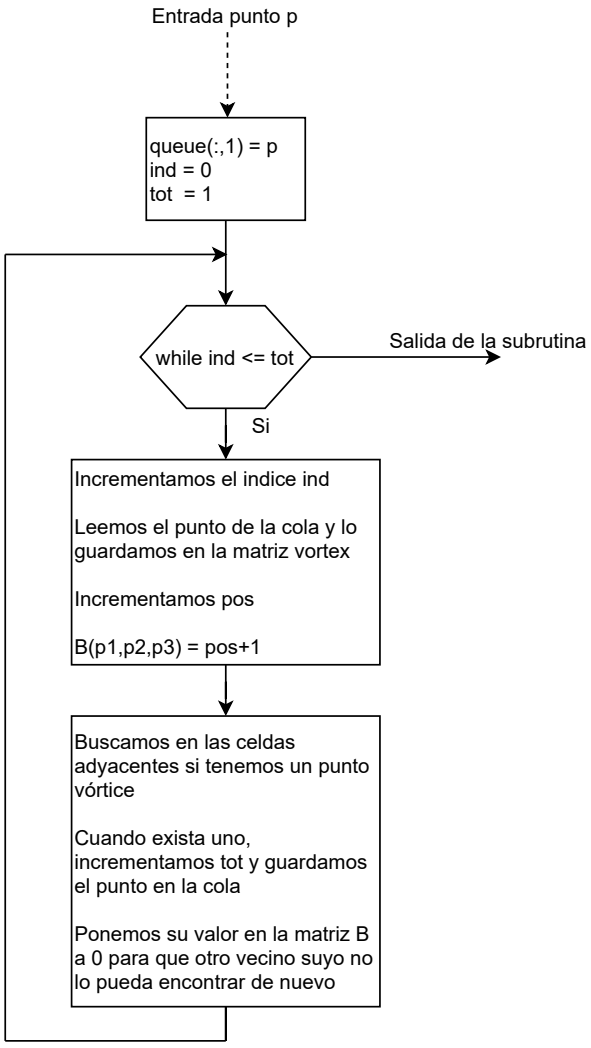


Figura 4.4: Diagrama de flujo de la subrutina de agregado de los vórtices

Como podemos ver en el diagrama de flujo, empezamos inicializando las variables de la cola y entramos en un bucle *while*. Lo primero que se hace es leer el punto que nos marca el índice `ind` para meterlo directamente a la matriz vórtice y guardamos en la matriz `B` la posición en la que se encuentra dicho punto en la matriz vórtice sumado una unidad. Si guardáramos la posición simplemente, el primer punto tendría posición 1, que se podría confundir un con 1 de la identificación, lo que haría que identificáramos el mismo punto dos veces. El guardar la posición que ocupa dentro de la matriz va a ser útil en el paso posterior de unificar los vórtices. Una ejemplo de como quedaría la matriz `B` tras este cambio lo mostramos en la siguiente ecuación.

$$\begin{bmatrix} 0 & 1 & 1 & 0 & 0 \\ 1 & 1 & 0 & 0 & 1 \\ 1 & 1 & 1 & 0 & 1 \\ 0 & 0 & 0 & 1 & 1 \\ 1 & 1 & 0 & 1 & 1 \end{bmatrix} \longrightarrow \begin{bmatrix} 0 & 210 & 211 & 0 & 0 \\ 212 & 213 & 0 & 0 & 218 \\ 214 & 215 & 216 & 0 & 219 \\ 0 & 0 & 0 & 220 & 221 \\ 225 & 226 & 0 & 222 & 223 \end{bmatrix} \tag{4.3}$$

Una vez ya está añadido el punto hay que buscar en las celdas adyacentes si estas contienen un 1, como se vio en la figura 4.2. En el caso de tratar con celdas interiores en el dominio no hay mayor problema y se mira cada una de las 6 direcciones, pero hay que prestar especial atención en el caso de tener una celda en el contorno. Si la celda se encuentra en un borde en la dirección y no se mira en dicha dirección ya que tenemos la pared. Si nos encontramos en la dirección x, aunque el dominio completo si que tiene simetría en esta dirección, al realizar la partición se tratará como una frontera sin simetría y simplemente no se mirará en dicha dirección, esto se tendrá en cuenta posteriormente. Para el caso de la dirección z si que hay que tener en cuenta la simetría y mirar en la celda adyacente en el lado opuesto del dominio.

Al encontrar un punto adyacente, lo añadimos a la cola aumentando también el índice `tot`, y ponemos el punto en la matriz `B` indicando que ya ha sido añadido a la cola y será procesado, evitando que otros puntos vecinos suyos lo vuelvan a meter, evitando puntos duplicados en el sistema de colas.

Podemos ampliar este algoritmo para encontrar estructuras distintas, como por ejemplo *sweeps* y *ejections*. En este caso la matriz `B` contendrá 0, 1 y 2, siendo *ejection* = 1 y *sweep* = 2. Aquí, cuando se encuentra el punto semilla se mira su identidad y se introduce además en la función de agregado, buscando únicamente conexiones con puntos que tengan la misma identidad. Se necesita un vector de identidades que nos dice la identidad de cada vórtice en la matriz de vórtices. Habría que realizar ligeras modificaciones, como por ejemplo hacer `B(p1,p2,p3) = pos + 2`, ya que ahora estamos buscando hasta los doses. Sin embargo, no es ninguna modificación del concepto del algoritmo y es fácilmente adaptable a encontrar el número de estructuras diferentes que se requiera.

Al final de este algoritmo cada procesador tiene sus propios vórtices, teniendo cada uno un tamaño diferente de matriz vórtice. Lo siguiente que se va a hacer es que los esclavos manden al procesador maestro sus matrices vórtice. Este las va a recibir y las va a ordenar

una debajo de la otra. Como hemos comentado, en la matriz `B` de cada procesador se ha guardado la posición en la que estaba almacenado dicho punto, pero ahora al ponerse todos sobre la misma matriz estas posiciones ya no coinciden. Como lo único que se ha hecho es un desfase de los mismos, se guarda un pequeño vector del tamaño del número de procesadores que estén calculando poniendo en su lugar correspondiente cual es el desfase que tienen los puntos de dicho procesador. Una vez que se tiene esto se realiza una difusión de este vector al resto de procesadores y pasamos al siguiente paso de la identificación.

4.2.2. Unificación de estructuras

En este apartado vamos a ver como se puede solucionar el problema de dividir un vórtice por el hecho de realizar la división del dominio. El caso más sencillo es el que se muestra en la figura 4.3, donde un vórtice en forma de hilo es separado por la mitad. Sin embargo, esto puede ser mucho más complicado, como por ejemplo un vórtice en forma de U roto de manera transversal, el cual generaría tres vórtices inicialmente que deben ser unificados en uno solo. Para este caso y casos todavía más complicados se ha ideado una solución.

El primer paso que vamos a realizar es enviar el último plano YZ al procesador siguiente al nuestro. El procesador 1 lo mandará al procesador 2, el 2 al 3, ..., y el último lo mandará al primero. Estamos ante una comunicación circular entre los procesadores que se muestra de manera gráfica en la figura 4.5. En este tipo de comunicaciones todos los procesadores tienen que enviar y recibir un plano, por lo que tienen que estar coordinados para que unos estén enviando y otros recibiendo. Lo que se hace para solucionar este problema es que todos los procesadores esperan a a recibir el plano antes de enviarlo, excepto uno de ellos. El procesador maestro manda primero su plano iniciando así la comunicación, y espera a recibir el plano que necesita, que lo recibirá cuando el resto de comunicaciones hayan terminado.

Cada procesador va a comparar su primer plano con el plano que ha recibido, que era el último de su procesador anterior. Para ello se va a comparar punto a punto, comprobando que en ambos planos tengan un valor mayor que cero. Si esto ocurre, tomamos la posición que está guardada y le aplicamos el desfase, guardándola en un vector de conexiones. El código empleado es el siguiente:

```
do jj = 1,ny
    do ii = 1,nz
        if (ownplane(ii,jj) > 0 .and. plane(ii,jj) > 0) then
            posOwn   = ownplane(ii,jj)+vecCont(myid+1)-1
            posPlane = plane(ii,jj)   +vecCont(myid)  -1
            vectorConex(1,posOwn)   = posPlane !1 left
            vectorConex(2,posPlane) = posOwn   !2 right
        endif
    enddo
enddo
```

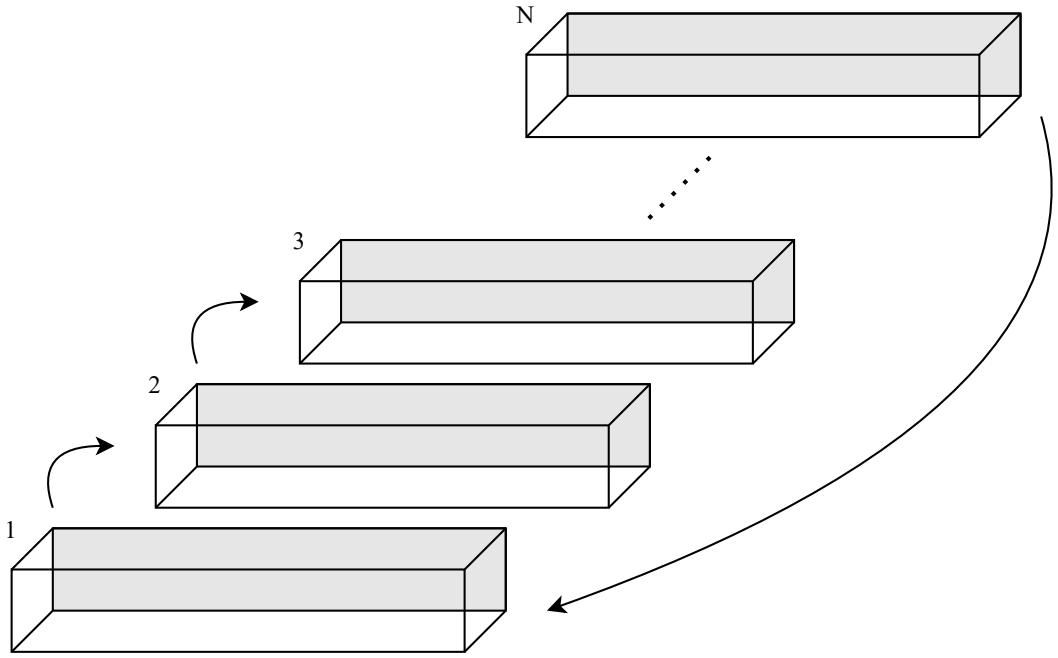


Figura 4.5: Visualización de una comunicación circular entre procesadores

Como podemos ver, al tomar la posición también se le resta 1 a la posición, y esto es debido a que, como ya se ha comentado, se había guardado en la matriz `B = pos+1`. El vector de conexiones tiene dos filas, una con las conexiones de un punto a izquierdas, y otra con las conexiones del punto a derechas, siendo a izquierdas ir hacia procesadores con un ID más pequeño y a derechas con un ID mayor. En este caso, hemos recibido el plano del procesador anterior, por lo que su punto se encontrará a izquierdas nuestro, y nuestro punto se encuentra a derechas suyo. Además, las conexiones son recíprocas, a izquierdas, en nuestra posición en el vector, guardamos la posición del punto del plano recibido, mientras que a derechas, en la posición del punto del plano recibido, guardamos nuestra posición.

Una vez que cada procesador ha hecho sus conexiones, se hace la suma de todos los procesadores mediante un `MPI_REDUCE`, teniendo el procesador maestro todas las conexiones, y ya se encuentra preparado para generar la matriz vórtice final.

La siguiente parte del código es realizado únicamente por el procesador maestro, mientras el resto de procesadores esperan. Este podría ser uno de los cuellos de botella de nuestro algoritmo, pero es lo suficientemente rápido para que no lo sea, por lo que se ha mantenido. Una futura ampliación sería pensar otra manera de unificar los vórtices realizada por varios procesadores a la vez.

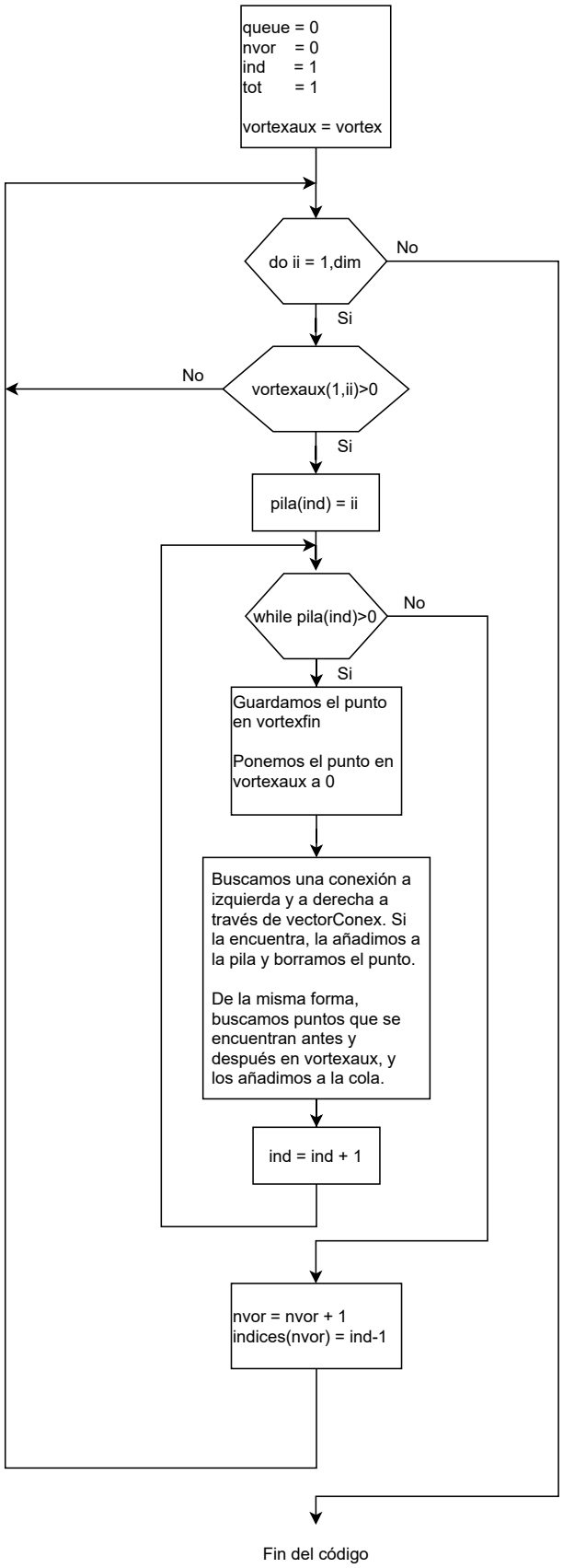


Figura 4.6: Diagrama de flujo del proceso de unificación de los vórtices

La idea que se muestra en la figura 4.6 tiene varias similitudes con la ya vista en el agregado de estructuras. Se utiliza un sistema de colas con el que vamos a ir añadiendo los puntos a la matriz `vortexfin`. Recordamos que en la actual matriz `vortex` los vórtices están separados por una columna de ceros, así que lo primero que hace el código es recorrer todos los puntos de la matriz vórtice auxiliar y buscar un punto que sea distinto de cero. Cuando lo encuentra mete su índice en la cola y entramos en el bucle `while`. Lo primero que hacemos es guardar el punto de la cola y lo borramos de la matriz vórtice auxiliar. Después buscamos todas las conexiones de dicho punto, primero la posición anterior y posterior del punto en la matriz vórtice, y después mirando sus conexiones a izquierda y derecha, haciendo uso del vector de conexiones que se ha generado anteriormente, metiendo los índices correspondientes a la cola y repitiendo este proceso.

Al final del diagrama de flujo podemos ver como se cambia la forma de identificar cada vórtice dentro de la matriz vórtice. La forma anterior tenía dos desventajas importantes: por un lado el aumento de memoria correspondiente al tener que dejar una columna vacía para poder separar un vórtice de otro, y por otro lado, si queremos acceder a un vórtice en concreto, tenemos que recorrer toda la matriz vórtice hasta llegar al vórtice que nos interesa. Ahora generamos un vector llamado `indices`, del tamaño del número de vórtices total, donde cada elemento es la posición en `vortex` en la que acaban los puntos de dicho vórtice. Por poner un ejemplo, queremos obtener los puntos del vórtice 5, el código para obtener el índice inicial y el índice final sería el siguiente:

```
indi = indices(5-1)+1
indf = indices(5)
```

Con estos dos índices, al hacer `vortex(:,indi:indf)` nos devolverá todos los puntos que contiene el vórtice 5. Existe un caso particular para el primer vórtice, y es que el índice inicial no se puede obtener de esta forma ya que nos vamos de límites del vector, en este caso directamente `indi = 1`. Esta forma de almacenar información es muy eficiente, ya que los datos están muy compactos en una única matriz y solo necesitamos un vector auxiliar de tamaño igual al número de vórtices, el cual ocupa muy poco, por lo que esta misma manera de almacenar se verá posteriormente en el algoritmo de seguimiento temporal.

Finalmente los cambios que hay que realizar para poder identificar distintos tipos de vórtices son muy pocos. El único relevante es a la hora de hacer la comparación entre planos, y es que ahora no nos vale que en la posición tenga un valor mayor que 0, ahora deben de cumplir que sean vórtices del mismo tipo. Lo que se hace en este caso es guardar estos planos antes de modificar la matriz `B` y trabajamos ahora con dos planos, el que teníamos anteriormente con las posiciones y el nuevo que tenemos formado por 0, 1 y 2, en el caso de los *sweeps* y *ejections*. La comprobación ahora es que en este último plano tengan valores mayores que cero y sean del mismo tipo, utilizando el plano de las posiciones para almacenar estos valores en el vector de conexiones. Una vez esta completo el vector conexiones, el resto del programa es exactamente el mismo, añadiendo el vector de identidades.

Una vez el procesador maestro tiene todos los vórtices con sus respectivos índices, los reparte entre los distintos procesadores para realizar el posterior filtrado y cálculo de las características. Para hacer el reparto se hace la siguiente división:

$$\texttt{div} = \frac{\texttt{nvor}}{\texttt{nummpi}} \tag{4.4}$$

En este caso la división no tiene que ser entera, pero al estar trabajando con números enteros, Fortran nos da la división sin el resto. Todos los procesadores menos el último tendrán un número de vórtices igual a `div`, mientras que el último tendrá estos más los que han quedado del resto de la división. Además, hay que mandar el vector de índices que se corresponda a cada procesador, y para dejarlo correctamente hay que restar el elemento anterior a la parte de la lista que quieres mandar.

$$\begin{bmatrix} \vdots \\ 1000 \\ \hline 1021 \\ 1058 \\ \vdots \end{bmatrix} \longrightarrow \begin{bmatrix} 21 \\ 58 \\ \vdots \end{bmatrix} \tag{4.5}$$

En el ejemplo de la ecuación (4.5), si queremos coger los vórtices que están por debajo de la línea, los nuevos índices que resultan salen tras haber restado el valor inmediatamente superior, en este caso 1000.

4.2.3. Filtrado y cálculo de características

Una vez ya tenemos todos los vórtices unificados y repartidos entre los procesadores, el paso posterior es filtrar aquellos no deseados y obtener sus características que nos permitirán hacer física de este problema. El filtrado de vórtices es esencial para eliminar todos aquellos vórtices que han podido nacer por el propio ruido de la simulación. Además, si tenemos vórtices muy pequeños es muy difícil de seguirlos temporalmente, por lo que tendríamos un gran número de vórtices apareciendo y desapareciendo. Los criterios que se han seguido para eliminar ciertos vórtices son:

- Eliminamos todos aquellos vórtices con un tamaño inferior a 7 puntos.
- Eliminamos los vórtices que tienen todos los puntos en una posición y constantes, ya que físicamente estos vórtices planos no tienen sentido.

El filtro de los 7 puntos puede parecer arbitrario, pero es mucho más pequeño que el que propone Lozano-Durán en [23], el cual elimina todos los vórtices menores a $V^+ = 30^3$

para evitar los problemas de resolución de la malla, y a pesar de que elimina un 70 % de los vórtices encontrados, suponen menos de un 1 % del volumen total de las estructuras, por lo que la energía contenida en estos es muy pequeña. En nuestro caso, aplicando el filtro que propone Lozano-Durán, nos eliminamos un 74 % de las estructuras y suponen un 1.2 % del volumen total, para el caso de identificación de las Qs. Para los vórtices mediante el método de Chong el número de estructuras que se eliminan es todavía mayor, cercano al 90 % del total, pero solo suponen un 1.5 % del volumen. Puesto que con este filtro el volumen que se elimina es muy bajo, con el filtro de 7 puntos que proponemos somos todavía menos restrictivos.

Para comprobar el filtro del tamaño, simplemente tomando los índices del vórtice podemos saber su tamaño y eliminarlo si es necesario. Para el filtro del vórtice plano tomamos todos los puntos del vórtice y los recorremos comprobando cuando vale el valor de y, si este valor cambia con respecto al valor que teníamos guardado ya no es vórtice plano, se acepta y se sale del bucle, pero si acabamos el bucle y esta valor sigue constante, el vórtice es plano, por lo que se elimina de la lista. Una vez tenemos un vector que nos indica que vórtices continúan y cuáles se eliminan volvemos a reconstruir la matriz vórtice y la matriz índice para cada procesador.

Una vez tenemos la lista definitiva de vórtices que se van a guardar en cada procesador, procedemos al cálculo de las características principales:

- **Volumen:** El volumen de una estructura lo calculamos como la suma del volumen de sus puntos individualmente. El volumen que posee una celda es el siguiente:

$$V_i = \Delta x \cdot \Delta z \cdot \Delta y^{(i)} \tag{4.6}$$

  Donde Δx y Δz son constantes para todo el dominio, y Δy depende de la altura en la que nos encontremos, y se puede calcular como se ha visto en la ecuación (4.1). El volumen de la estructura coherente es por tanto $V = \sum V_i$.

- **Centro de masas:** la definición matemática de centro de masas es la siguiente:

$$\boldsymbol{r}_{cm} = \frac{\sum_i V_i \boldsymbol{r}_i}{\sum_i V_i} = \frac{1}{V} \sum_i V_i \boldsymbol{r}_i \tag{4.7}$$

  Aquí hay que diferenciar en el caso en el que exista simetría o no exista. En el caso de que sea un vórtice interior, el cálculo se realiza directamente tomando el volumen individual de cada celda multiplicándolo por su vector posición, hacer la suma de todas las celdas y dividiéndolo entre en el volumen total del vórtice.

  En el caso de ser un vórtice con simetría la forma de calcularlo cambia. En este caso, se supone que un vórtice no llega a ser lo suficientemente grande para estar dividido por la simetría y además superar la mitad del dominio. En la dirección en la que exista simetría, desplazamos los puntos que se encuentren en la primera mitad del dominio al final sumándole N_x o N_z dependiendo de cual sea la simetría y calculamos

el centro de masas. Al terminar comprobamos el valor, si el centro de masas está dentro del dominio el cálculo ha sido correcto, pero si ha caído fuera de este hay que restarle L_x o L_z para que se encuentre dentro del dominio. Esto se ilustra en la figura 4.7, en el que en este caso particular el centro de masas queda fuera del dominio por tener más masa en la primera mitad del dominio que la segunda, y es por esto que habría que restar L_x a la primera dimensión de r_{cm}.

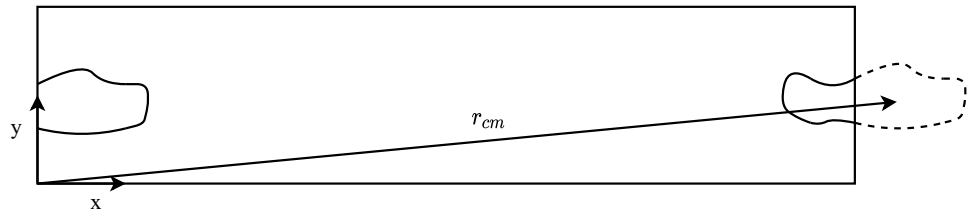


Figura 4.7: Cálculo del centro de masas de un vórtice con simetría

- **Caja envolvente:** la caja envolvente de una estructura coherente es la caja con aristas paralelas a los ejes cartesianos más pequeña que engloba todos los puntos de nuestro vórtice. En este caso también hay que diferenciar si estamos trabajando con simetría o sin simetría. Cuando estamos tratando con una estructura interior, la caja se puede calcular directamente como al diferencia entre la dimensión máxima y la mínima en cada dirección.

  En el caso en el que alguna dirección tenga simetría se busca la dimensión máxima en la primera mitad del dominio (max_1) y la dimensión mínima en la segunda mitad (min_2). Una vez lo tenemos, la caja se puede calcular como (suponiendo la dirección x):

$$\Delta_x = L_x + max_1 - min_2 \tag{4.8}$$

  Esto se puede ver gráficamente en la siguiente figura:

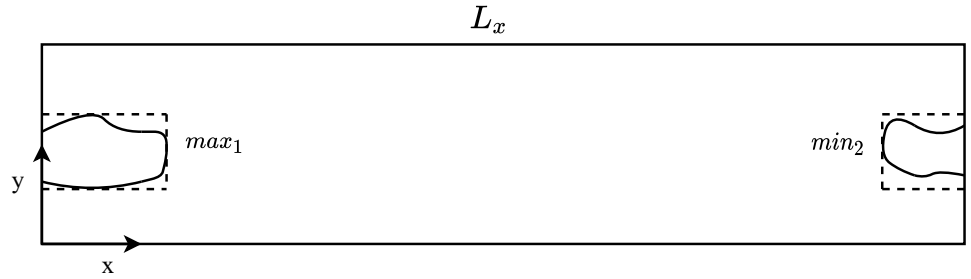


Figura 4.8: Caja envolvente a una estructura coherente

- **Posición del vórtice:** para poder situar el vórtice dentro del dominio, guardamos el punto que es la posición mínima del vórtice en todas las dimensiones. Esta variable recibe el nombre de `bound` y se calcula de forma diferente si tenemos un vórtice con periodicidad o un vórtice interior. Si estamos ante un vórtice interior, `bound` se

calcula como la posición mínima del vórtice en cada dimensión, que coincide con el vértice de la caja envolvente más cercano al origen de coordenadas.

En el caso que una dimensión tenga simetría, tomamos la dimensión min_2 vista en la caja envolvente y le restamos L_x o L_z en función de la dirección que tenga la simetría. Esto lo que hace es trasladar la parte del vórtice que se encuentra al final del dominio justo antes de que empiece, lo que da valores de `bound` negativos. Esto queda representado en la figura 4.9. Sabiendo cual es el punto `bound` y cuales son las dimensiones de la caja, el vórtice queda totalmente localizado espacialmente.

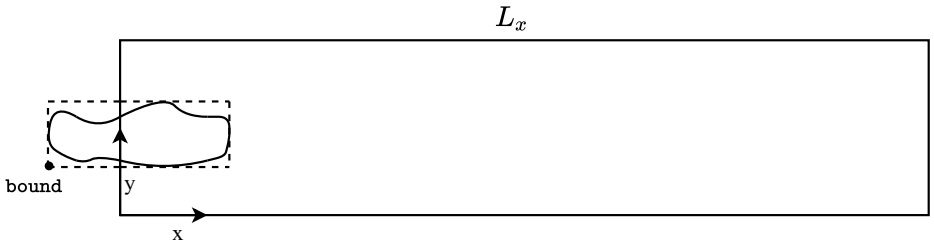


Figura 4.9: Posición mínima del vórtice sobre el dominio con simetría

4.3. Validación y resultados

Una vez tenemos desarrollado el algoritmo, el siguiente paso es comprobar que los resultados sean correctos, además de que la idea conceptual sea correcta. En este caso la validación se ha hecho por dos métodos: por un lado hemos creado un archivo con una matriz booleana de calibración, en la que se han puesto distintos vórtices topológicamente diferentes para ver si el algoritmo es capaz de identificarlos correctamente, y por otro lado observando los vórtices que vamos obteniendo del código DNS, verificando en casos concretos que el vórtice tiene sentido y las características están calculadas correctamente.

Nuestro archivo de calibración consta de los siguientes vórtices:

- Vórtice con simetría en x, verificando que la construcción del vector conexiones es correcta y la unificación de los vórtices también, mostrado en la figura 4.10a.
- Vórtice con simetría en z realizada por un único procesador, mostrado en la figura 4.10b.
- Vórtice de grandes dimensiones en forma de U, el cuál ocupa casi toda la dimensión x, por lo que va a ser identificado por diferentes procesadores y luego debe ser unificado correctamente, mostrado en la figura 4.10c.
- Vórtice con doble simetría en x y en z, mostrado en la figura 4.10d.

- Vórtice pequeño situado diagonalmente a otro más grande, el cuál no tiene conexión debido a que no hay conexiones en direcciones diagonales, mostrado en la figura 4.10e.
- Vórtice con dos núcleos claramente marcados, unidos entre sí mediante un hilo de vórtice, lo que hace que en global sea un único vórtice, mostrado en la figura 4.10f.
- Dos vórtices, cada uno situado en una pared del canal, que no deben dar el mismo puesto que no existe periodicidad en esta dirección, mostrado en la figura 4.10g.

Como podemos ver en la figura 4.10, algunos de los vórtices que se han generado no tienen sentido físico, pero nos sirve para verificar que nuestro algoritmo funciona correctamente sean cuales sean las características de los vórtices, tengan o no sentido físico. Con esto ya tenemos el algoritmo preparado para ser lanzado en cualquier Reynolds de fricción que deseemos.

Lo importante del algoritmo que hemos desarrollado es, además de ser funcional, que sea lo suficientemente rápido y que la disminución del tiempo sea importante a la hora de aumentar el número de procesadores, para poder implementarlo en un superordenador. A continuación, mostramos una tabla de tiempos para una matriz booleana tanto para $Re_\tau = 500$ como para $Re_\tau = 1000$, usando distinto número de procesadores.

N° Procesadores	Tiempo para $Re_\tau = 500$ [s]	Tiempo para $Re_\tau = 1000$ [s]
2	15.55	119.83
4	9.19	72.06
8	6.44	55.20
16	5.77	49.89
32	5.44	48.60
64	5.36	48.04

Tabla 4.2: Tabla de tiempos en segundos del algoritmo de identificación

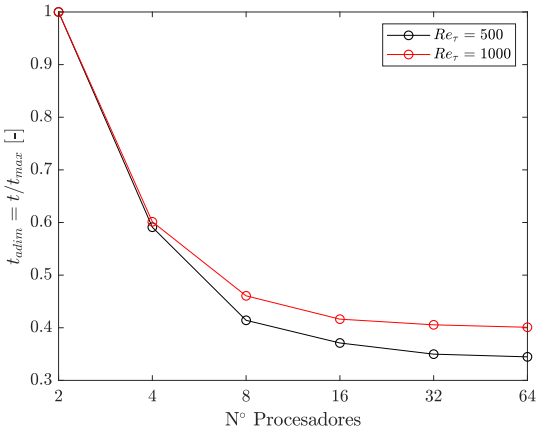
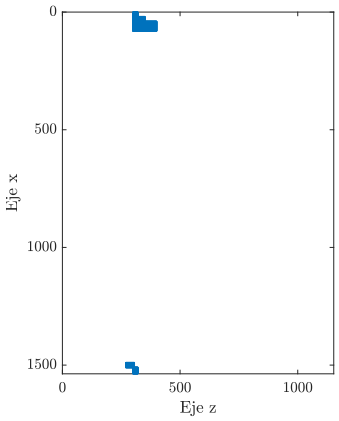
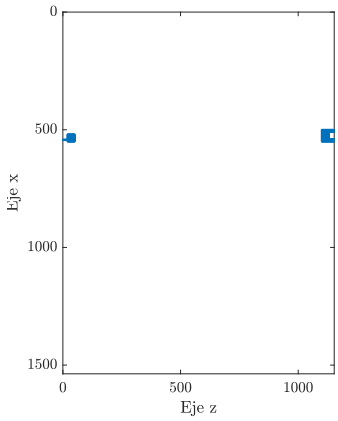


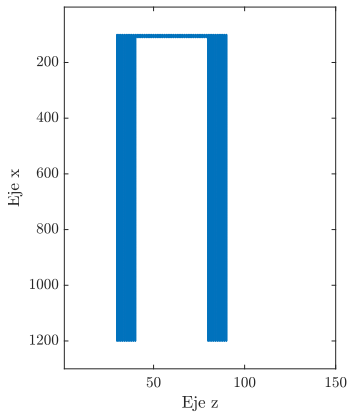
Figura 4.11: Visualización de la reducción de tiempo adimensional en el algoritmo de identificación al incrementar el número de procesadores



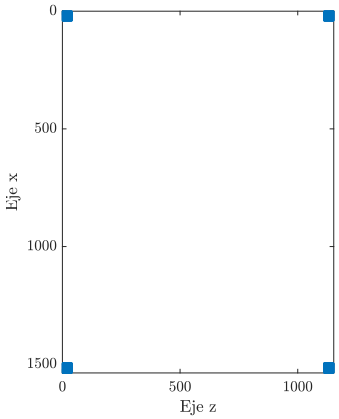
(a) Vórtice con simetría en x



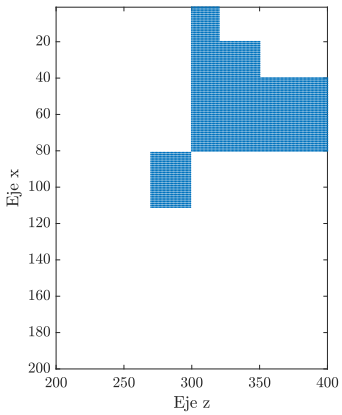
(b) Vórtice con simetría en z



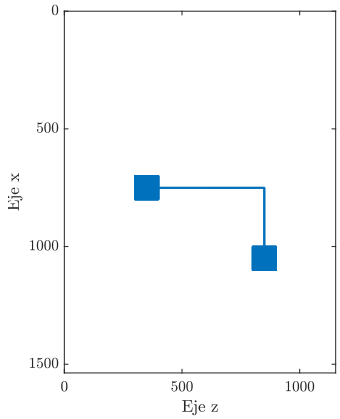
(c) Vórtice con forma de U



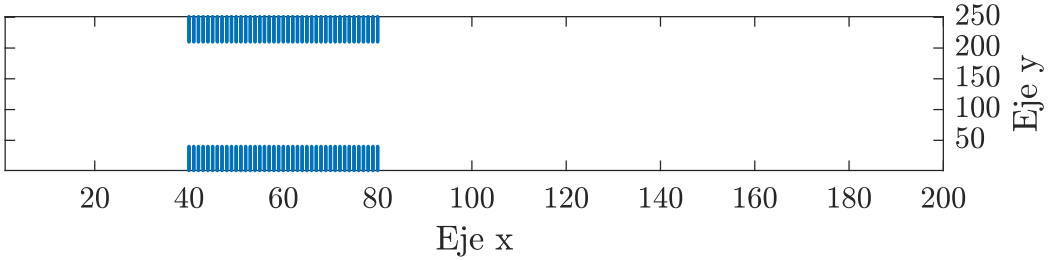
(d) Vórtice con dos tipos de simetría



(e) Vórtice pequeño cercano a grande



(f) Vórtice con dos núcleos unidos por un hilo



(g) Dos vórtices, cada uno en una pared

Figura 4.10: Visualización gráfica de los vórtices de calibración

El resultado del tiempo se ha obtenido tras ejecutar 10 veces el algoritmo y hacer la media. La figura 4.11 muestra el resultado del tiempo adimensionalizado por el tiempo máximo en cada caso, que se produce con 2 procesadores. Como podemos ver en ambos casos, el mayor beneficio se obtiene cuando tenemos pocos procesadores y los aumentamos, mientras que si ya estamos en un número alto de procesadores y lo aumentamos todavía mas, la mejora que obtenemos es muy pequeña. Esto quiere decir que llega un punto en el que la tarea paralelizada ya ha sido optimizada lo máximo que se podía y el tiempo se invierte en tareas que no están paralelizadas, como por ejemplo la lectura del archivo o la unificación de vórtices. Además el aumentar el número de procesadores lleva consigo un aumento del número de comunicaciones, lo que puede llevar a que la mejora no sea tan importante.

La velocidad nos interesa que sea lo suficientemente alta para no ralentizar al código LISO, que como veremos en el capítulo 6, este algoritmo se ejecutará en paralelo junto con el de DNS. Para tener un orden de magnitud, el código DNS para $Re_\tau = 500$ y 32 procesadores tarda unos 8 segundos por paso temporal, y genera un campo booleano cada 5 instantes temporales, lo que da un campo booleano cada 40 segundos. En la tabla 4.2 podemos ver que con cualquier número de procesadores podemos ser más rápidos que el LISO, por lo que podemos determinar que nuestro algoritmo es lo suficientemente rápido en calcular los vórtices que posteriormente se utilizarán en el seguimiento temporal.

A continuación vamos a mostrar unas imágenes de dos vórtices, uno identificado mediante el criterio de Chong y otro mediante la identificación de eventos uv.

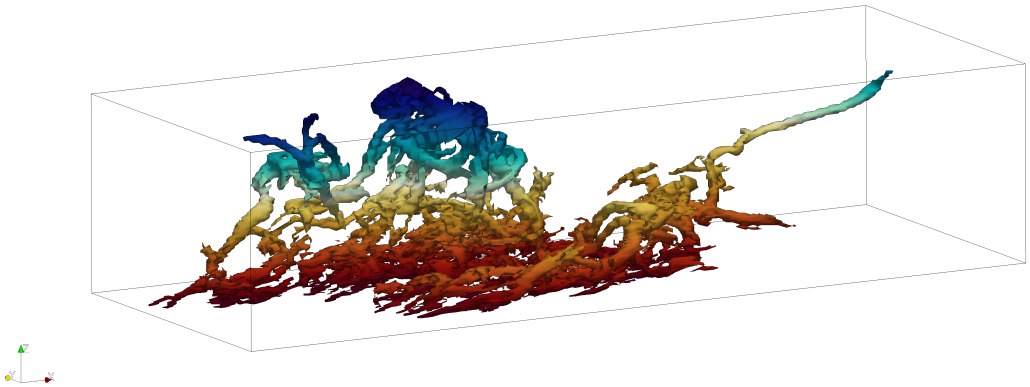


Figura 4.12: Vórtice identificado mediante el criterio de Chong

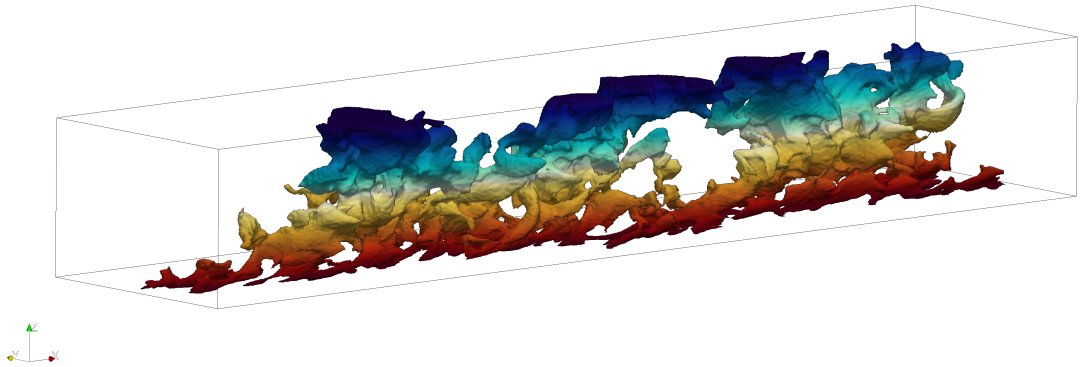


Figura 4.13: Vórtice identificado mediante eventos *uv*

En ambos casos la identificación de los vórtices parece correcta, ya que la forma es coherente y no existen discontinuidades entre distintas partes. Esto junto a la correcta identificación de los vórtices en la calibración nos da una idea de que el código es correcto.

Además, la forma de los vórtices, ya comentada en [23] coincide con la obtenida. Para los vórtices identificados por el criterio de Chong, o *vortex cluster*, se caracterizan por tener forma de "esponja de gusanos" formado por una gran cantidad de hilos unidas con unos diámetros del orden de 7η, mientras que los vórtices identificados por eventos *uv* tienen forma de "esponja de copos', siendo estructuras muy alargadas en la dirección del flujo.

Capítulo 5

Seguimiento temporal de estructuras coherentes

El principal objetivo de este capítulo es obtener un algoritmo que nos permita conocer cuál es la evolución temporal de un vórtice y obtener todas sus características en cada instante temporal. Como resultado final tendremos una base de datos con todos los vórtices y las interacciones que surgen entre ellos. Esto nos permitirá conocer las interacciones que existen entre diferentes tipos de vórtices en función de la distancia a la pared, y como se integra dentro de los mecanismos de la turbulencia.

En las siguientes secciones se describirá el método de seguimiento de los vórtices y la gestión de la memoria y de la información para que sea eficiente, además de explicar como se ha realizado la paralelización. Cabe recordar que el objetivo es implementar el algoritmo de identificación y seguimiento junto con el código DNS para que se calculen de forma simultánea.

A lo largo de este capitulo se hará una descripción del problema que queremos abordar, se explicarán los algoritmos de seguimiento y almacenamiento de datos y finalmente se mostrarán una serie de resultados.

5.1. Descripción del problema

El punto de partida de este algoritmo es la identificación de vórtices que se ha realizado en el capítulo anterior. Ese algoritmo tenia por salida una serie de matrices y vectores que nos indicaban los puntos que contenían cada vórtice, su volumen, su centro de masas, la caja envolvente y la posición de la caja envolvente, lo que nos permitía tener caracterizados por completo los vórtices. Para cada instante temporal tendremos un archivo con las características de los vórtices, independientes entre si. Este archivo lo guardamos con extensión ".vor" y el algoritmo de seguimiento debe leerlos de forma seguida para tener

los instantes temporales ordenados. El archivo final que genera el algoritmo de seguimiento tendrá la extensión ".time".

En el cálculo DNS podemos establecer cada cuantos pasos temporales guardamos un campo booleano. Este valor no debe ser excesivamente grande ya que los campos no se parecerían entre sí y el seguimiento se complicaría, ni tampoco excesivamente pequeño ya que generaría gran cantidad de archivos y de datos, ralentizando el cálculo y ocupando mucha memoria. En nuestro caso se genera una nueva matriz cada 5 pasos temporales, el cual es suficiente para poder seguir tanto las estructuras pequeñas como grandes del flujo. El archivo ".vor" tiene además una serie de información relativa a tamaño de la malla y el instante temporal en el que se ha generado el archivo.

La idea general de este algoritmo es que toma dos instantes consecutivos, compara los vórtices y es capaz de determinar el nacimiento, muerte, evolución, rotura o coalescencia de ellos. Para ello tomamos dos instantes, uno t_n y otro t_{n+1} y generaremos un vector para cada uno de los instantes temporales del tamaño del número de vórtices que existan en cada instante, a los que llamaremos `conex1` y `conex2` respectivamente. En función del número que contenga en cada elemento del vector nos dirá que ocurre con dicho vórtice. El funcionamiento de estos vectores es fundamental ya que todo el algoritmo gira en torno a ellos, y su funcionamiento es el siguiente.

- **Evolución:** un vórtice en t_n continua en otro vórtice de t_{n+1}. Para que esto ocurra el vórtice en `conex1` debe de tener guardada la posición del vórtice en el siguiente instante temporal, en valor positivo. De la misma forma, el vórtice final en `conex2` debe tener guardada la posición de su vórtice de origen también en valor positivo.

- **Nacimiento:** todos los vórtices que nacen se especifican en el instante t_{n+1}, es decir, en `conex2`. Todos los vórtices que tengan un valor 0 o negativo son vórtices que nacen nuevos. Si tienen un 0 significa que han nacido espontáneamente y no provienen de ningún otro vórtice (en principio, ya se verán las uniones), y si tienen un valor negativo quiere decir que es un vórtice que ha nacido por separarse de otro vórtice, indicando la posición en negativo del vórtice del que proviene.

- **Muerte:** la especificación es similar a los vórtices de nacimiento pero ocurren en el instante t_n, en `conex1`. Los vórtices que tengan un valor 0 o negativo son vórtices que mueren. Si tienen un 0 significa que el vórtice ha muerto espontáneamente por disipación (en principio, ver las roturas), y si tiene un valor negativo es que ha muerto por unirse a otro vórtice, indicando la posición del vórtice al que se une con valor negativo.

- **Rotura:** el criterio de rotura es arbitrario, pero en nuestro caso consideraremos que un vórtice se ha roto si ninguno de lo vórtices resultantes tiene un volumen de al menos $0.5V_i$. Para indicar una rotura en nuestros vectores, el vórtice que se ha roto en `conex1` tiene un valor de 0, como si hubiera muerto espontáneamente, mientras que todos los vórtices resultantes en `conex2` tienen conexión negativa indicando el vórtice de la rotura.

- **Unión:** de forma análoga a las roturas, consideramos que exite una unión cuando dos o más vórtices se unen y ninguno de ellos tiene al menos el $0.5V_i$ de volumen del vórtice resultante. En este caso todos los vórtices iniciales tienen conexión negativa en `conex1` indicando la posición del vórtice resultante, mientras que en `conex2` el vórtice resultante tiene un valor de 0, como si hubiera nacido espontáneamente.

A continuación mostramos un ejemplo de dos vectores `conex1` y `conex2` para ver una aplicación de todos los conceptos que acabamos de ver.

$$\texttt{conex1} = \begin{bmatrix} 2 \\ 0 \\ \text{-4} \\ \text{-4} \\ 0 \end{bmatrix} \qquad \texttt{conex2} = \begin{bmatrix} \text{-2} \\ 1 \\ \text{-2} \\ 0 \\ \text{-1} \end{bmatrix} \tag{5.1}$$

El ejemplo mostrado en la ecuación (5.1) es muy sencillo pero contiene gran cantidad de elementos que ya hemos explicado. Tenemos un vórtice que evoluciona, que es el vórtice 1 de `conex1` con el vórtice 2 de `conex2`. Tenemos una rotura del vórtice 2 de `conex1` que resulta en los vórtices 1 y 3 de `conex2`. También existe una unión de los vórtices 3 y 4 de `conex1` que mueren uniéndose al vórtice 4 de `conex2`, y finalmente el vórtice 5 de `conex1` muere espontáneamente y el vórtice 5 de `conex2` nace como un hijo del vórtice 1 del primer paso temporal. Podemos ver como a partir de dos vectores somos capaces de reconstruir la evolución de los vórtices de forma detallada, y son usados por la base de datos para actualizar los vórtices.

Como ya se ha comentado, el algoritmo consta de dos partes claramente diferencias: por un lado tenemos la comparación de estructuras y la generación de los vectores de conexiones, que se realiza mediante la comparación de dos instantes temporal, y por otro lado es la gestión de la memoria y la actualización de la base de datos, la cual debe ser eficiente y aprovechar al máximo el uso de memoria.

- **Comparación de estructuras:** esta parte del código es la que se encarga directamente de hacer la comparación entre dos instantes temporales y crear los vectores conexiones que servirán para actualizar la base de datos. Se basa en la idea de que la mayor parte de los vórtices van a ser muy similares a los del instante temporal siguiente, por lo que gran parte de ellos se identificarán de forma muy rápida, mientras que para el resto se comprobarán las intersecciones entre vórtices y este proceso será mas lento.

- **Base de datos:** el principal problema que concierne a la base de datos es que van a convivir vórtices vivos junto con vórtices que ya hayan muerto. Los vórtices vivos van a seguir actualizándose y aumentando el tamaño de sus vectores mientras que los vórtices muertos seguirán como están. Esto debe ser solucionado de una forma

eficiente, manteniendo los datos del mismo vórtice contiguos en memoria y evitar guardar memoria innecesaria.

Finalmente tenemos que mencionar como se va a realizar la paralelización en este caso. Para este algoritmo vamos a plantear una paralelización híbrida entre MPI y OpenMP. Empezando por la paralelización MPI tenemos una configuración *master-slave*, en la que cada esclavo se va a encargar de coger dos instantes temporales y va a generar los vectores `conex1` y `conex2` que se usarán para actualizar la base de datos, mientras que el procesador maestro se va a dedicar únicamente a la base de datos, recibiendo los vectores de conexión de los esclavos y actualizando la estructura. La paralelización por memoria compartida se va a realizar dentro de cada esclavo, y cada procesador se va a encargar de calcular un elemento en concreto de los vectores conexión, siendo fácilmente paralelizable por OpenMP ya que escriben en zonas de memoria distintas.

5.2. Comparación de estructuras

Esta sección se centra en la explicación del algoritmo que siguen los procesadores esclavos a la hora de comparar dos instantes temporales. Los instantes temporales son consecutivos, es decir, el primer procesador tendrá los instantes 1 y 2, el segundo los instantes 2 y 3, y así sucesivamente. El algoritmo se basa en la idea de que los campos en dos instantes de tiempo sucesivos no cambian excesivamente, y la mayoría de los vórtices habrán sufrido una ligera modificación de posición y de volumen, como se muestra en la figura 5.1.

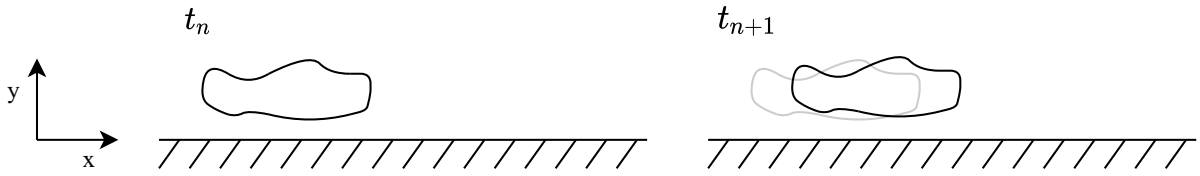


Figura 5.1: Esquema representativo de la evolución de un vórtice en dos instantes temporales planteados en dos instantes temporales. El dibujo de la izquierda se corresponde con el instante temporal t_n y el de la derecha con el instante t_{n+1}

Los vórtices que son como el mostrado en la figura 5.1 se identifican con el método que llamamos conexiones *fáciles*, en las que el vórtice apenas ha cambiado y tiene un volumen similar, con centros de masas cercanos y características similares. Cuando un vórtice no es tan fácil de identificar pasamos a las denominadas conexiones *difíciles*, en la que las características ya no son tan similares y tenemos que buscar la intersección de las cajas de los vórtices para buscar las conexiones, lo que lleva más tiempo en evaluar. Finalmente, se ha desarrollado un último filtro que analiza todos aquellos vórtices que no han tenido conexión y busca de manera más exhaustiva si tienen algún tipo de relación con algún otro vórtice, o simplemente son nacimientos o muertes espontáneas.

El esquema de OpenMP que se sigue en toda la función es muy sencillo y es el mostrado en la figura 5.2. Como podemos ver, tenemos una región paralela que se abre y no se cierra hasta el final de la función, y entre medias tenemos los distintos filtros que se basan en recorrer todos los vórtices con un bucle *DO*. Esta figura es una simplificación, ya que entre los filtros se ha usado `OMP SINGLE` cuando era necesario inicializar variables comunes a todos los procesadores.

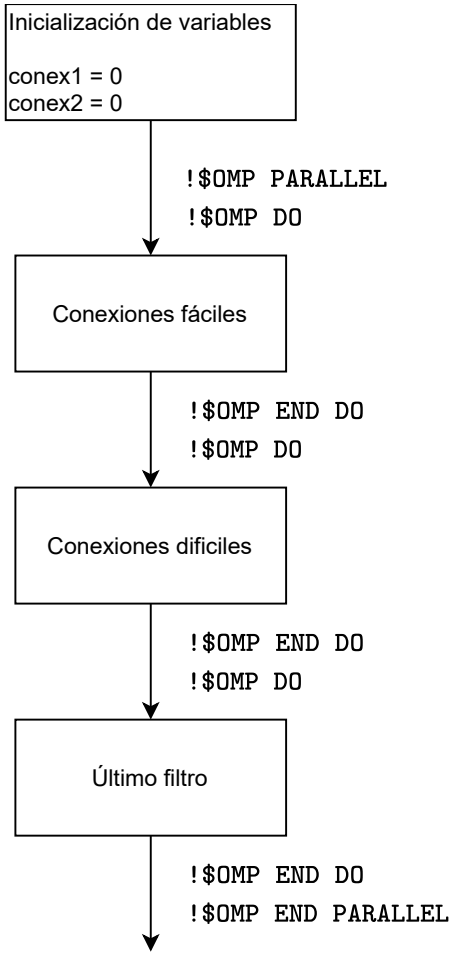


Figura 5.2: Esquema de la estructura principal del algoritmo de comparación de estructuras

Para el caso en el que estemos identificando distinto tipo de estructuras, como en el caso de identificación de las Qs, el algoritmo que vamos a mostrar es exactamente igual, lo que cambia es que en la identificación hay que meter una condición adicional para que la conexión sea entre estructuras del mismo tipo.

Con esto, pasamos a explicar en profundidad los diferentes tipos de conexiones y como se realiza la identificación en cada uno de ellos.

5.2.1. Conexiones *fáciles*

Las conexiones *fáciles* se encargan de calcular una distancia ficticia de cada vórtice en el instante t_n con todos los vórtices del instante t_{n+1}, obteniendo el valor más bajo. Si este valor es menor a un cierto umbral, la conexión se admite como fácil y se escribe en los vectores `conex1` y `conex2`.

El proceso que sigue el programa para encontrar las conexiones fáciles es el siguiente:

1. Una función calcula cual es la mínima distancia desde nuestro vórtice en t_n al vórtice más cercano en t_{n+1} y cuál es ese vórtice. Esta distancia ficticia se calcula a partir de la distancia de los centros de masas ponderada con el resto de características geométricas, teniendo en cuenta la simetría del dominio.
2. Comprobamos que el error mínimo de dicha estructura sea menor que el valor umbral determinado mediante una calibración.
3. Mediante el uso de la orden `!$OMP CRITICAL` obligamos a los procesadores a pasar individualmente por la zona en la que se comprueba que el vórtice con el mínimo resultado no haya sido encontrado anteriormente en el vector conexión. En el caso de que no haya sido encontrado, se guarda en la posición correspondiente. En el caso de que ya estuviera en el vector conexión, lo que se hace es comprobar la intersección punto a punto de ambos vórtices con el vórtice destino, y aquel que tenga un mayor número de conexiones es el que se queda con la conexión directa.

Para calcular la distancia ficticia lo que se hace es calcular la distancia entre los centros de masas y modificarla con las distintas características geométricas de la siguiente forma:

$$dist = k_V \cdot k_{\Delta_x} \cdot k_{\Delta_y} \cdot k_{\Delta_z} \cdot \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2 + (z_1 - z_2)^2} \tag{5.2}$$

Donde (x, y, z) se corresponde a las coordenadas cartesianas del centro de masas de cada vórtice que se esta comparando. Calcular esta distancia directamente tiene el problema del factor de escala, y resulta de que x puede tomar valores desde $x \in [0, 8\pi]$, mientras que $y \in [0, 2]$, por lo que las distancias en la dirección perpendicular a la pared van a tener muy poca relevancia respecto a las distancias en la dirección del flujo. Para ello se han corregido todas las distancias de los centros de masas para que en todos los casos se cumpla que $\{x, y, z\} \in [0, 1]$. Además, esta distancia no se calcula directamente como indica la ecuación (5.2), sino que se calcula también la distancia a través de la pared de simetría para las coordenadas x y z, y nos quedamos con la menor de ellas. Finalmente, las constantes modificadoras de la distancia se calculan de la siguiente forma.

$$k_V = \frac{V_1}{V_2} \qquad k_{\Delta_x} = \frac{\Delta_{x1}}{\Delta_{x2}} \qquad k_{\Delta_y} = \frac{\Delta_{y1}}{\Delta_{y2}} \qquad k_{\Delta_z} = \frac{\Delta_{z1}}{\Delta_{z2}} \tag{5.3}$$

Donde Δ_x, Δ_y, Δ_z son la longitud de la caja envolvente en cada dirección cartesiana. Los factores no son exactamente los mostrados en (5.3), sino que se hace que $k = \max(k, k^{-1})$ para que todas las constantes sean igual o mayor que 1. Esto hace que los vórtices que sean muy parecidos entre sí tenga unas constantes muy cercanas a la unidad y por lo tanto la distancia entre los centros de masas se vea muy poco modificada, mientras que si tenemos dos vórtices muy dispares pero muy cercanos, de esta forma hacemos que su distancia ficticia sea lo suficientemente grande para que no pase el umbral y no se tome como conexión fácil.

Finalmente, presentamos el diagrama de flujo del programa de las conexiones fáciles.

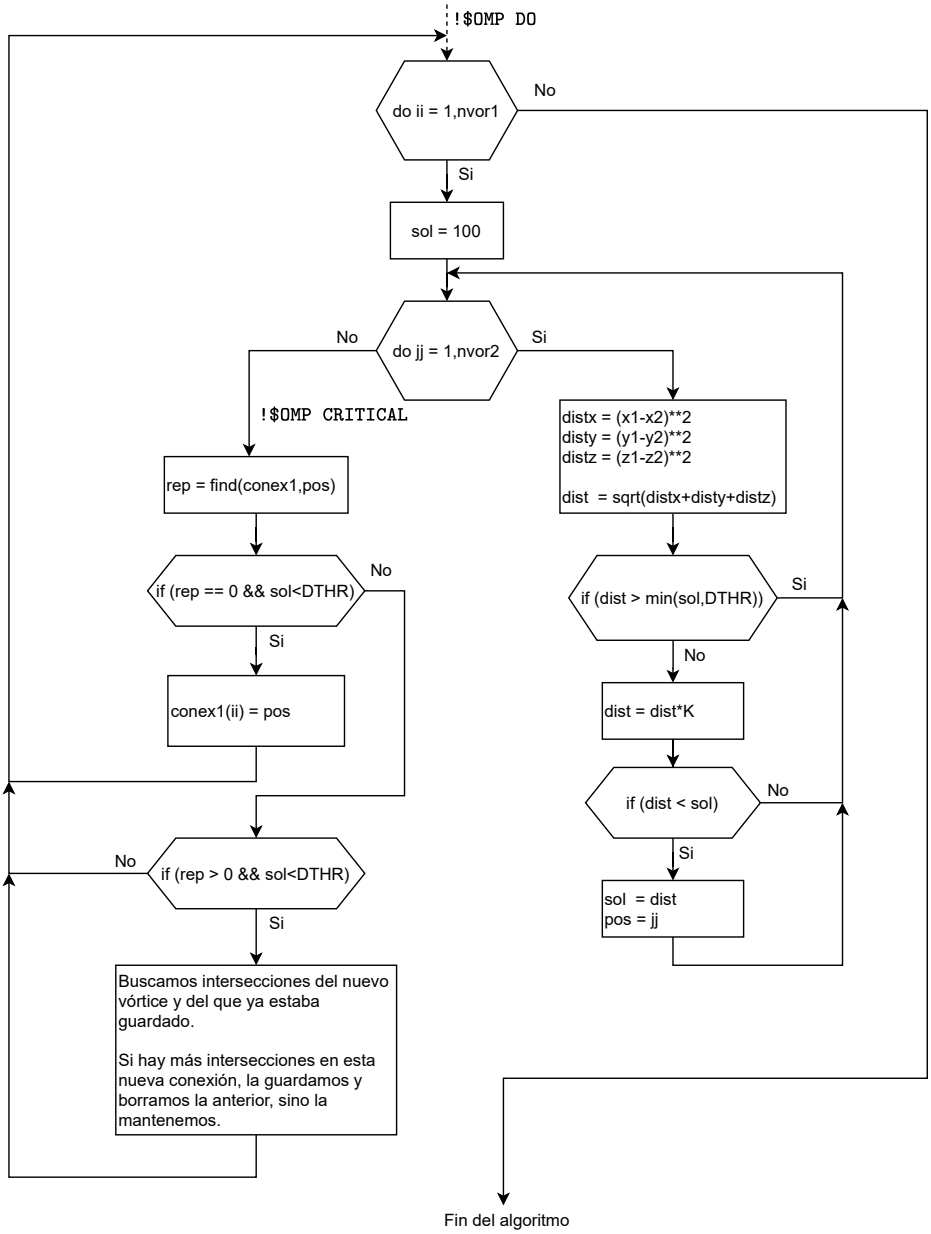


Figura 5.3: Diagrama de flujo del algoritmo seguido para la identificación de las conexiones *fáciles*

Desde el punto de vista de la paralelización por memoria compartida, no hay ningún problema ya que cada procesador va a estar trabajando en su propio elemento `ii` del vector `conex1`, ya que el resto de variables son privadas para cada procesador. Además, la información de los vórtices en el instante t_{n+1} no se modifican, únicamente se leen para sacar la distancia, por lo que no hay ningún problema desde este punto de vista. La inclusión de la región de `!$OMP CRITICAL` es esencial porque puede ocurrir que dos procesadores que hayan obtenido el mismo vórtice como solución busquen a la vez en el vector, por lo que ninguno encontraría ninguna repetición y ambos escribirían la misma conexión, cosa que no puede ocurrir según la idea de los vectores conexiones.

Finalmente, una vez tenemos todas las conexiones directas escritas en `conex1` se escriben todas las conexiones recíprocas en `conex2`.

5.2.2. Conexiones *difíciles*

Las conexiones difíciles comprueban todos aquellos vórtices que no han podido ser identificado por las características geométricas vistas en las conexiones fáciles, ya sea porque el criterio del umbral es muy estricto o porque no hay ningún vórtice cerca con forma similar. Es por esto que nace la necesidad de desarrollar un nuevo algoritmo que se encargue de obtener las conexiones directas de aquellos vórtices que no han podido ser identificados por el otro criterio. Además, va a encontrar los hijos, que son vórtices que nacen a partir de una separación de un vórtice más grande, y las uniones, que son vórtices que mueren por unirse a vórtices más grandes.

La forma de identificación de estas conexiones se hace a partir de la intersección tridimensional de las cajas en dos pasos temporales consecutivos. Se toma la caja del vórtice en cuestión, se genera una matriz tridimensional del tamaño de la caja y se buscan las intersecciones de los vórtices en el otro instante temporal dentro de ese dominio. Cuando hemos acabado tenemos una matriz tridimensional y en cada posición tenemos el índice del vórtice que ha hecho intersección en ese punto del dominio. A partir del estudio de esta matriz podemos determinar si existe una conexión directa, o estamos ante un hijo o una unión. El proceso que se sigue es el siguiente:

1. En primer lugar calculamos la posición del punto medio de la caja para todos los vórtices de ambos instantes temporales.

2. Recorremos toda la matriz de vórtices buscando un vórtice que no tenga conexión. Cuando lo encontramos (lo llamaremos vórtice objetivo), lo comparamos con todos los vórtices del otro instante temporal, tengan o no tengan conexión ya asignada. Se comprueba en primer lugar que la resta entre las posiciones de los centros de las cajas sea menor que la suma del semilado de cada caja en cada dirección. Esto se muestra en la figura 5.4, en la que podemos ver que en dirección x la resta de $\texttt{half2}_x - \texttt{half1}_x$ es mayor que la suma del semilado de ambas cajas $\Delta_x/2$, mientras que en dirección y la resta es menor y por tanto cumpliría el criterio de intersección

en esta dirección. Para que exista intersección real tiene que cumplirse en las tres direcciones cartesianas. En el caso que esto no se cumpla, el vórtice no se considera y se pasa al siguiente.

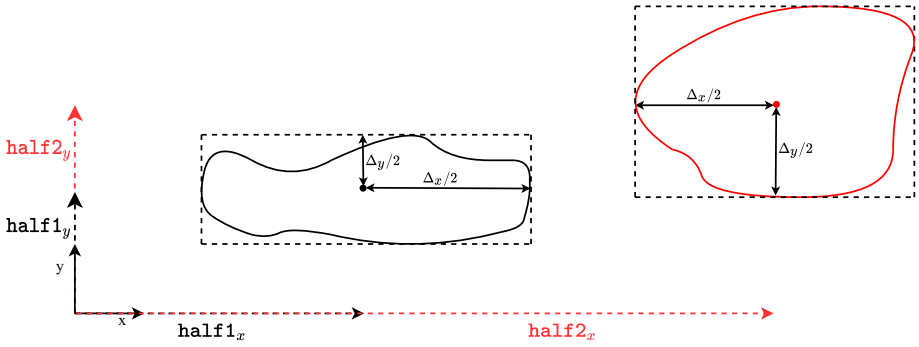


Figura 5.4: Boceto de la posición de los centros de las cajas de dos vórtices para conocer si intersecan

3. Una vez tenemos un vórtice que cumple el criterio por intersección de ambas cajas (lo llamaremos vórtice candidato), se recorre todos los puntos del vórtice candidato para ver si dicho punto se encuentra dentro de la caja del vórtice objetivo, guardando en el elemento donde hay coincidencia el índice del vórtice candidato. Aquí se ha tenido en cuenta que las coordenadas x de los puntos se han desplazado de media la velocidad de advención, por lo que teniendo la velocidad media del flujo $u_0(y)$ y la diferencia de tiempo Δt, podemos calcular cuanto se ha desplazado el vórtice en dirección x. Un ejemplo de como quedaría la matriz tridimensional tras haber hecho esto con todos los vórtices se muestra en la ecuación (5.4), en un caso bidimensional de ejemplo. Como podemos ver, tiene coincidencias tanto con los vórtices candidatos 5 y 14, teniendo mayor número de coincidencias con el candidato número 14.

$$\texttt{Rebuild} = \begin{bmatrix} 0 & 0 & 5 & 5 & 5 \\ 14 & 0 & 0 & 5 & 5 \\ 14 & 14 & 0 & 0 & 0 \\ 14 & 14 & 14 & 0 & 0 \\ 14 & 14 & 14 & 0 & 0 \end{bmatrix} \tag{5.4}$$

4. Después recorremos la matriz tridimensional y hacemos una lista con los candidatos y cuantas repeticiones tiene cada uno de los candidatos. Por la naturaleza de las conexiones difíciles, la mayor parte de conexiones van a ser de hijos que se separan de un vórtice grande o de uniones de un vórtice en otro, por lo que generamos un parámetro, llamado `RTHR` el cual puede tomar un valor entre 0 y 1. La idea es que normalmente de todos los candidatos nos interesa quedarnos con el que más volumen tenga, que es la conexión más probable, pero con el parámetro podemos ajustar cuanta importancia damos al número de repeticiones y cuanta importancia le damos al volumen más grande. Si le damos mucha importancia a las repeticiones solo en casos con un número de intersecciones similar cogerá aquel vórtice más grande, mientras que si hace caso al volumen, en el momento que un vórtice grande tenga

una sola intersección con nuestro vórtice será elegido. En nuestro caso lo hemos configurado a un valor `RTHR` = 0.8 dando prioridad a las repeticiones.

5. Una vez tengamos el vórtice candidato escogido, se guarda en la posición del vórtice objetivo con una conexión negativa

Este proceso se hace tanto para el instante temporal t_n como para el instante temporal t_{n+1}, rellenando los vectores de conexiones con conexiones negativas. Finalmente se hace búsqueda de conexiones recíprocas para buscar las conexiones directas que no han sido captadas por las conexiones fáciles, buscar la rotura de vórtices y la unión de varios vórtices similares en uno grande. Se recorren todas las conexiones negativas en el instante t_n, cuando se encuentra una se comprueba que sea recíproca, es decir, que el vórtice en t_{n+1} tenga guardado el índice del vórtice en t_n pero negativa. Si esto se cumple, se comprueba si en t_n hay mas conexiones negativas con el vórtice en t_{n+1} y se comprueba si en t_{n+1} hay más conexiones negativas con el vórtice en t_n. Se pueden dar tres casos:

- Puede ocurrir que la conexión recíproca sea la única que exista y no se haya encontrado ninguna conexión negativa más, por lo que hacemos la conexión directa. En el caso de que existan otras conexiones negativas, se hace uso de un nuevo parámetro, y es para discernir si un vórtice ha tenido un hijo o directamente se ha roto. En este trabajo se ha considerado que si uno de los vórtices resultantes tiene al menos la mitad del volumen del vórtice original se considera continuación y el vórtice no muere, continua en dicho vórtice.

- Otra posibilidad es que tengamos varios vórtices en t_n conectando con uno en t_{n+1} y ninguno de los vórtices tiene al menos la mitad de volumen del vórtice resultante, por lo que la conexión en `conex2` se pone a 0 y se considera una unión.

- La ultima posibilidad es que tengamos varios vórtices en t_{n+1} conectando con uno en t_n y que ninguno de los resultantes tenga al menos la mitad del vórtice original, por lo que la conexión en `conex1` se pone a 0 y se considera rotura del vórtice.

Desde el punto de vista paralelo de OpenMP, este algoritmo es perfectamente paralelizable, debido a que la mayor parte de variables son privadas. La única variable pública es el vector conexiones, y cada procesador va a trabajar sobre un vórtice a la vez, por lo que no existe ningún riesgo de que dos procesadores vayan a escribir en la misma posición de memoria.

Finalmente presentamos el diagrama de flujo mediante el cual se hace la identificación de las conexiones difíciles.

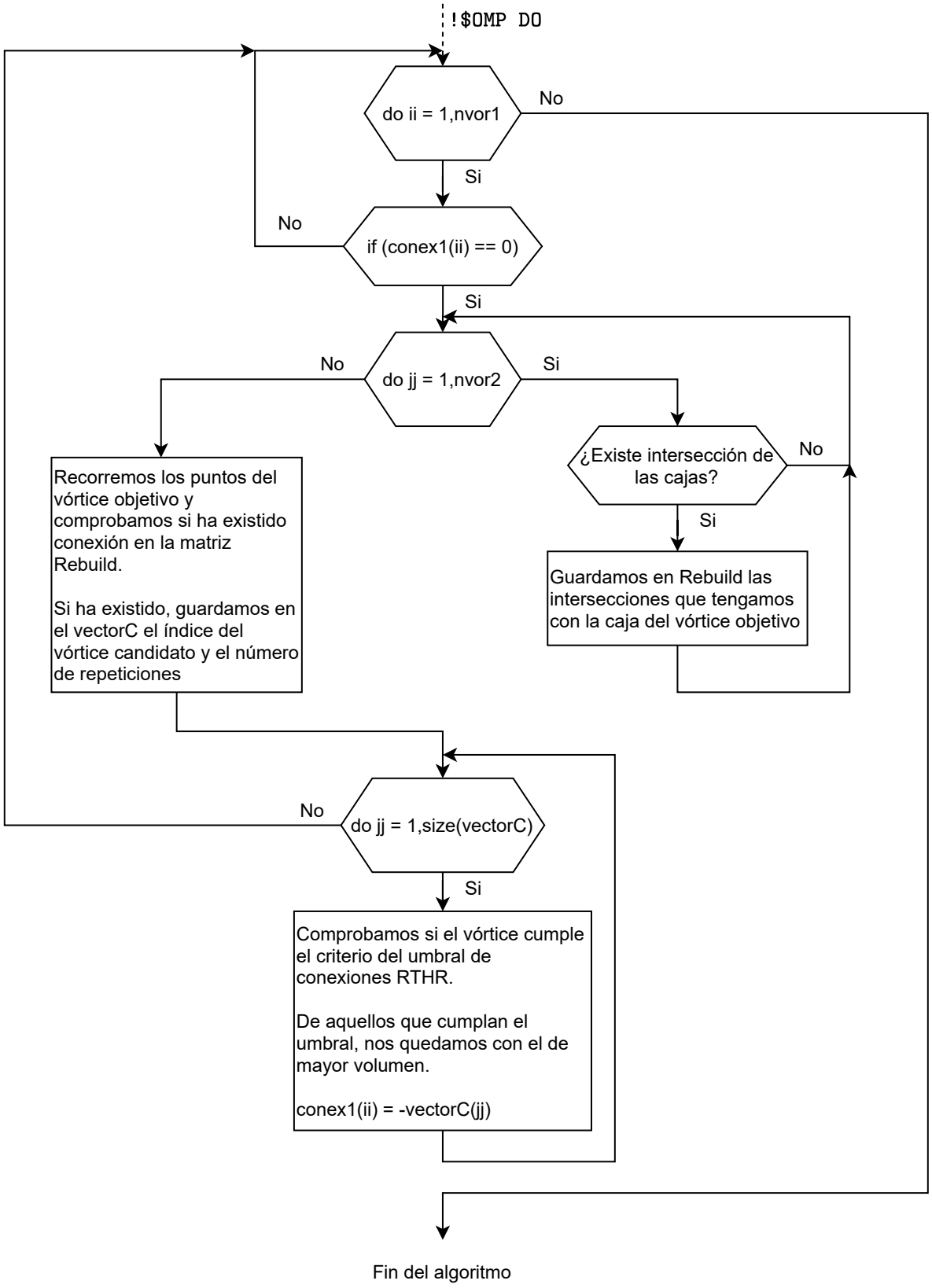


Figura 5.5: Diagrama de flujo del algoritmo seguido para la identificación de las conexiones *difíciles*

5.2.3. Último filtro

En este apartado vamos a explicar un último filtro que se ha desarrollado para terminar de captar todos aquellos vórtices que todavía siguen sin asignar. Este filtro se puede compilar o no mediante una macro a la hora de compilación por lo que se puede omitir, ya que como se verá apenas añade conexiones nuevas y por su naturaleza es bastante lento comparado a los anteriores. En este punto ya se ha captado la conexión de todos los vórtices grandes y la mayoría de las conexiones de los vórtices pequeños, pero puede ocurrir que vórtices muy pequeños se hayan desplazado más que lo que indica la velocidad de advección por lo que no hemos captado la conexión directa, o hijos de un vórtice más grande se hayan alejado tanto que no han podido ser reconocidos como hijos.

El proceso que se siguen en este filtro es el siguiente:

1. Buscamos un vórtice objetivo que no tenga ningún tipo de conexión.

2. Al igual que en las conexiones difíciles, buscamos aquellos vórtices candidatos que hacen intersección con nuestro vórtice objetivo. En este caso, a diferencia de las conexiones difíciles, hacemos que la caja del vórtice objetivo sea ligeramente más grande para poder captar vórtices que antes no hacían intersección por poco. El aumento ficticio del tamaño de la caja viene dado por un parámetro que se ha tomado $\texttt{SR} = 3$ unidades de malla en cada dirección en este caso.

3. Una vez tenemos un vórtice candidato, calculamos la distancia superficial más cercana entre ambos vórtices. Para ello se calcula la distancia entre todos los puntos del vórtice objetivo con todos los puntos del vórtice candidato, obteniendo el mínimo de todos ellos.

4. Si la distancia es menor que un cierto umbral, inicialmente establecido como $\texttt{DTHR} = 2$ unidades de malla, comprobamos la conexión del vórtice candidato. Si el vórtice candidato ya tenía una conexión directa, buscamos todos aquellos vórtices que tienen una condición indirecta con él. Por ejemplo, si el vórtice objetivo se encuentra en t_{n+1} y el vórtice candidato en t_n, buscamos todos los vórtices en el instante t_{n+1} que tienen conexión con el vórtice candidato y sumamos sus volúmenes, para comprobar si el volumen que estamos añadiendo a los vórtices resultantes puede ser albergado. En el caso de que pueda albergar el volumen que estamos añadiendo, lo ponemos como un hijo suyo.

5. En el caso que el vórtice candidato no tenga ninguna conexión, comprobamos la diferencia de volumen entre ambos vórtices. Si esta diferencia es menos de la mitad entre ellos, lo consideramos conexión directa, mientras que si no es así no se considera la conexión.

6. Si se ha guardado conexión, se guarda el valor de la distancia superficial obtenida como nuevo umbral y se siguen comprobando el resto de vórtices para ver si encontramos un candidato más idóneo.

Este proceso se realiza tanto en el instante temporal t_n como en el instante t_{n+1}. Como hemos visto, este filtro es mucho más tedioso y tiene gran cantidad de parámetros. La elección de estos parámetros, aunque coherentes, hace que la identificación pueda parecer arbitraria y con menor significado físico. Sin embargo, se ha comprobado que las conexiones que establece tienen sentido y no representan un porcentaje importante del número total de conexiones, ya que la gran mayoría ya son identificados por las conexiones fáciles y difíciles.

El hecho de tener que calcular la distancia de todos los puntos del vórtice objetivo con todos los puntos del vórtice candidato, para todos los vórtices candidatos, hace que este filtro sea el más lento de todos. Debido al gran coste computacional y el poco porcentaje de identificación, en caso de ser necesario puede ser eliminado a la hora de compilar el programa.

Finalmente, el diagrama de flujo relativo al proceso seguido en este último filtro se presenta en la figura 5.6.

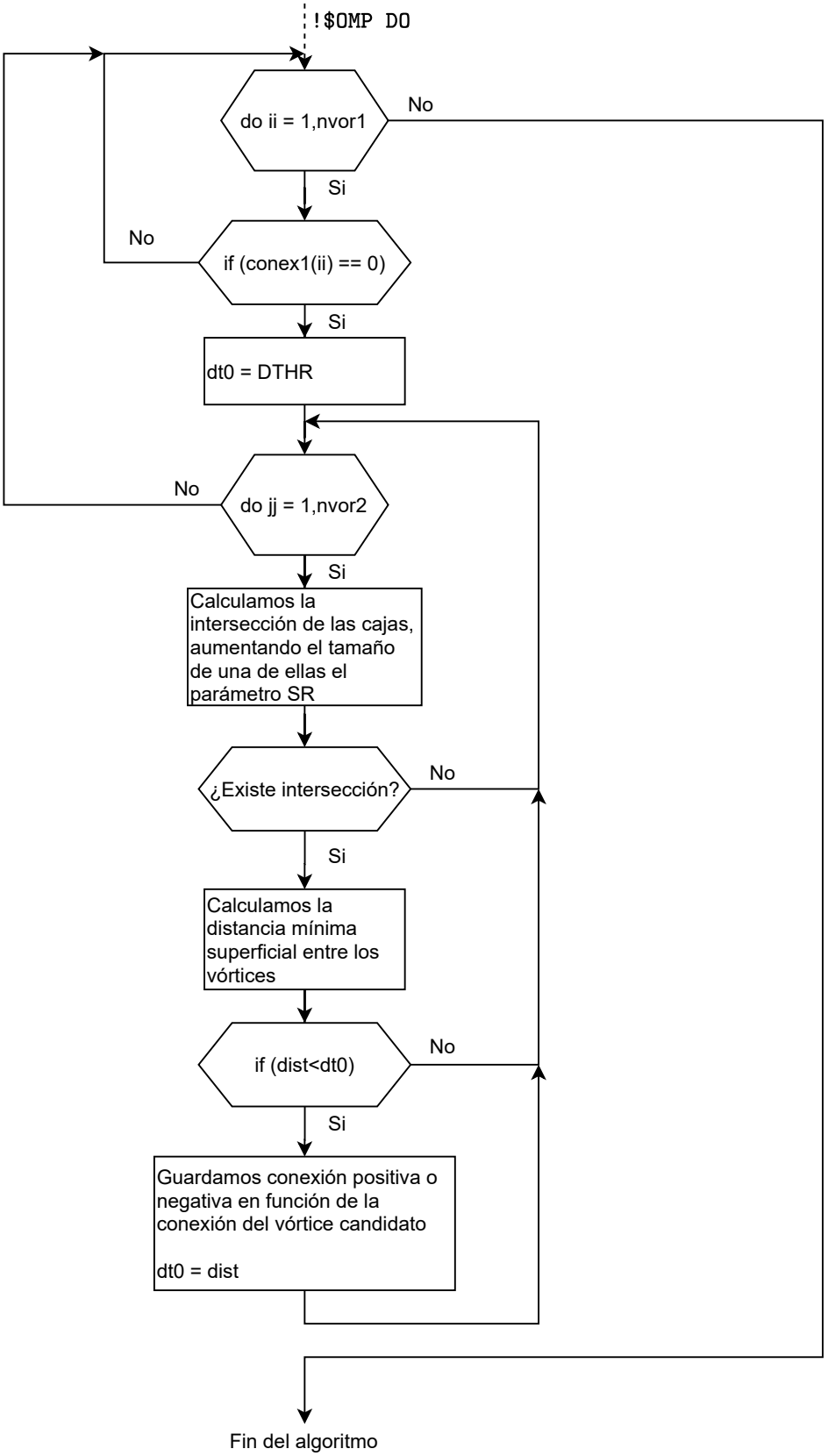


Figura 5.6: Diagrama de flujo del algoritmo seguido para el último filtro

5.3. Base de datos

La gestión de datos y como se va a administrar la base de datos de los vórtices es algo fundamental para el correcto funcionamiento del algoritmo y para tratar los resultados de una manera sencilla. La gran problemática de este apartado es que en la base de datos van a convivir vórtices vivos con vórtices ya muertos y esto supone que cada vórtice va a vivir distinto número de instantes temporales. Vamos a guardar variables que dependen de los instantes temporales, como por ejemplo la posición del centro de gravedad en cada instante, por lo que la dimensión de dicho vector no es igual para cada vórtice. Se han ideado tres formas de resolver este problema, y son las siguientes:

- Hacer uso de tipos derivados, que son tipos de datos similares a las estructuras presentes en MATLAB. Esto nos permite generar un vector individualmente para cada vórtice, lo que permite hacer un buen uso de la memoria y simplifica la legibilidad de la base de datos. Sin embargo, los principales inconvenientes es que los datos de los distintos vórtices no están relativos en memoria, lo que ralentiza el código, y a la hora de guardar los datos en HDF5 deberíamos crear un dataset para cada variable de cada vórtice, lo que consume una gran cantidad de tiempo.
- La segunda solución consiste en unificar todos los vectores en una única matriz, donde cada columna corresponde con cada vórtice y el número de filas viene dado por el máximo tamaño del vector más grande. Si un vórtice ha vivido menos instantes temporales, el resto de datos de la matriz se ponen a 0. Esto permite que los datos estén relativos en memoria y sea fácil acceder a ellos, pero conforme aumentan el número de vórtices cada vez las matrices estarán mas vacías, lo que ocupará mucha memoria innecesaria.
- La última solución es similar a la anterior pero en este caso colocamos todos los vectores seguidos en un vector más grande. Para poder acceder a los datos de cada vector hay que hacer uso de índices, de la misma forma que ya hemos visto en la identificación. Esta solución es la que se ha adoptado finalmente debido a que usa la memoria justa y los datos están contiguos en memoria, a pesar de que ser más complicado trabajar de esta forma.

La base de datos consta de las siguientes variables, que nos van a permitir obtener toda la información temporal de los vórtices:

- `index`: este vector contiene un índice que se le asigna al vórtice cuando nace. Tiene un tamaño igual al número de vórtices y suele coincidir su valor con la posición en el vector.
- `createstep`: vector de tamaño igual al número de vórtices que indica en qué paso temporal ha nacido.

- `endstep`: vector de tamaño igual al número de vórtices que indica en que paso temporal muere. Si todavía no ha muerto el vórtice contiene un 0.
- `split`: vector de tamaño igual al número de vórtices que nos indica si el vórtice en cuestión ha nacido por ser un hijo de otro vórtice, siendo el valor de este vector el índice del vórtice del que nos hemos separado.
- `join`: vector análogo al anterior pero en este caso indica si el vórtice en cuestión ha muerto por unirse a otro vórtice más grande, indicando en este el índice del vórtice al que nos hemos unido.
- `nsteps`: vector de tamaño igual al número de vórtices que nos indica cuantos instantes temporales ha vivido el vórtice.
- `nchildren`: vector de tamaño igual al número de vórtices que nos indica cuantos hijos ha tenido nuestro vórtice.
- `npartner`: vector de tamaño igual al número de vórtices que nos indica cuando vórtices se han unido al nuestro.
- `timecreate` y `timeend`: vectores de tamaño igual al número de vórtices que nos indica los tiempos de simulación en los que nace y muere el vórtice respectivamente.
- `iden`: vector del tamaño igual al número de vórtices que se usa en el caso de estar identificando Qs, y se utiliza para identificar si estamos ante un *sweep* o un *ejection*.
- `nstepsacum`, `nchildrenacum` y `npartneracum`: vectores de tamaño igual al número de vórtices que son los índices para poder localizar nuestro vórtice en cuestión dentro de las matrices y vectores que vamos a ver a continuación. El sistema de índices es exactamente igual al ya visto en el apartado de identificación.
- `V`: vector en el que se guarda el volumen de todos los vórtices para todos sus instantes temporales. El tamaño de este vector es igual al número de vórtices por el número de instantes temporales que vive cada vórtice. Para acceder a un instante determinado de cada vórtice, se hace uso del vector de índices `nstepsacum`.
- `CDM`: matriz que contiene la posición del centro de gravedad de cada vórtice en cada instante temporal. Es una matriz de 3 $\times$ número de vórtices por número de instantes temporales que vive cada vórtice, y para acceder a esta se hace uso del vector `nstepsacum`.
- `strvorstep`: vector que contiene el índice local que tiene el vórtice en los archivos ".vor" procedentes de la identificación, para cada instante temporal. El tamaño de este vector es igual al número de vórtices por el número de instantes temporales que vive cada vórtice. Para acceder a un instante determinado de cada vórtice, se hace uso del vector de índices `nstepsacum`.
- `border`: matriz que contiene la posición máxima y mínima del vórtice en cada dirección cartesiana y para todos los instantes temporales. Al igual que se vio con la variable `bound`, en el caso de vórtices con simetría, las posiciones mínimas pueden

tomar valores negativos. Está compuesta por x_{min}, x_{max}, z_{min}, z_{max}, y_{min} y y_{max}, en este orden, lo que nos da información de la caja y la localización en el espacio del vórtice. Es por esto que tiene una dimensión de 6 × número de vórtices por número de instantes temporales que vive cada vórtice, y para acceder a esta se hace uso del vector `nstepsacum`.

- `children`: vector que contiene los hijos que ha tenido cada vórtice. En este caso existe la posibilidad de que un vórtice no contenga ningún hijo, por lo que el tamaño de este vector no depende del número de vórtices. Para la lectura de este vector se comprueba si el vórtice tiene algún hijo con al variable `nchildren` y si existe el hijo ya se mira el índice en `nchildrenacum`.
- `partner`: vector que contiene los vórtices que se han unido a otros vórtices. Al igual que en el caso de los hijos, el número de uniones es indeterminado y no depende del número de vórtices. La lectura de las uniones se hace de la misma forma que con los hijos, pero usando `npartner` y `npartneracum` respectivamente.

Para identificar que un vórtice se ha roto o un vórtice ha nacido de una unión de varios vórtices más pequeños se realiza de una forma similar a la ya vista en los vectores de conexiones. Para identificar un vórtice que se ha roto en varios tenemos que buscar aquellos con `join` = 0, como si hubiera muerto de forma espontánea, y que tengan hijos que hayan nacido en el instante temporal siguiente a la muerte del vórtice de rotura. En el caso de una unión la idea es similar, pero en este caso tenemos `split` = 0 y uniones que han muerto en el instante temporal anterior a nuestro nacimiento.

La base de datos se inicializa con el primer paso temporal de manera directa, y a partir de aquí se tiene que ir actualizando conforme le llegan los vectores conexión procedentes de los procesadores esclavos. El proceso de actualización consta de las siguientes etapas:

1. En primer lugar se actualizan los vórtices que ya existen en la lista. Recorremos toda la base de datos y miramos si el vórtice ya estaba muerto o estaba vivo. Si se encontraba muerto se copian todos sus datos directamente. Si se encontraba vivo se mira que ocurre con el vórtice en el vector conexión, ya que puede morir en este instante, ya sea de manera espontánea o uniéndose a otro vórtice, o puede continuar, y en ese caso añadimos la nueva posición a `V`, `CDM`, `strvorstep` y `border`.
2. Se cuentan los nuevos vórtices y ampliamos la base de datos. Después añadimos los nuevos vórtices al final.
3. Al actualizar la base de datos guardamos en `split` y `join` las conexiones negativas que hemos recibido de los vectores conexión. Sin embargo, estos índices hacen referencia a las posiciones de los vórtices en sus respectivos instantes temporales, pero nos interesa conocer el índice dentro de la base de datos. Se ha generado una función que se encarga de buscar dicho vórtice usando el vector `strvorstep` y actualizamos los valores de `split` y `join` con los nuevos índices, además de añadir los hijos a `children` y las uniones a `partner` con los índices correctos.

Un problema que puede ocurrir es que la base de datos se haga demasiado grande por tener muchos vórtices muertos que vamos a tener que seguir recorriendo, además de llenar demasiado la memoria RAM. Es por esto que lo que hace el algoritmo es eliminar todos los vórtices muertos tras haber pasado un número de instantes temporales que podemos fijar. El programa genera un archivo temporal que almacena los vórtices muertos, y al final del programa lo que realiza es una lectura de dichos archivos temporales y unifica toda la base de datos, generando un único archivo de salida.

Uno de los puntos negativos de esta forma de gestionar la memoria es que no podemos paralelizarla mediante memoria compartida, es decir, OpenMP. Esto es porque a la hora de montar las matrices con datos de todos los instantes temporales, no sabemos a priori en qué posición se va a situar cada vórtice, sino que se tiene que ir haciendo de uno en uno. Sin embargo, este no es un gran problema, ya que la actualización de la estructura es lo suficientemente rápida eliminando los vórtices muertos de esta.

Mostramos en la figura 5.7 el diagrama de flujo del proceso de actualización de la base de datos de manera simplificada.

Finalmente, cuando se ha hecho el seguimiento de todos los pasos temporales se tiene que hacer la unificación de todos los vórtices. El código lee todos los archivos temporales de vórtices muertos que ha ido creando y monta la base de datos completa, ordenando los vórtices en función de su índice. Una vez tenemos toda la base de datos creada, la escribe a disco en un archivo HDF5 con extensión ".time".

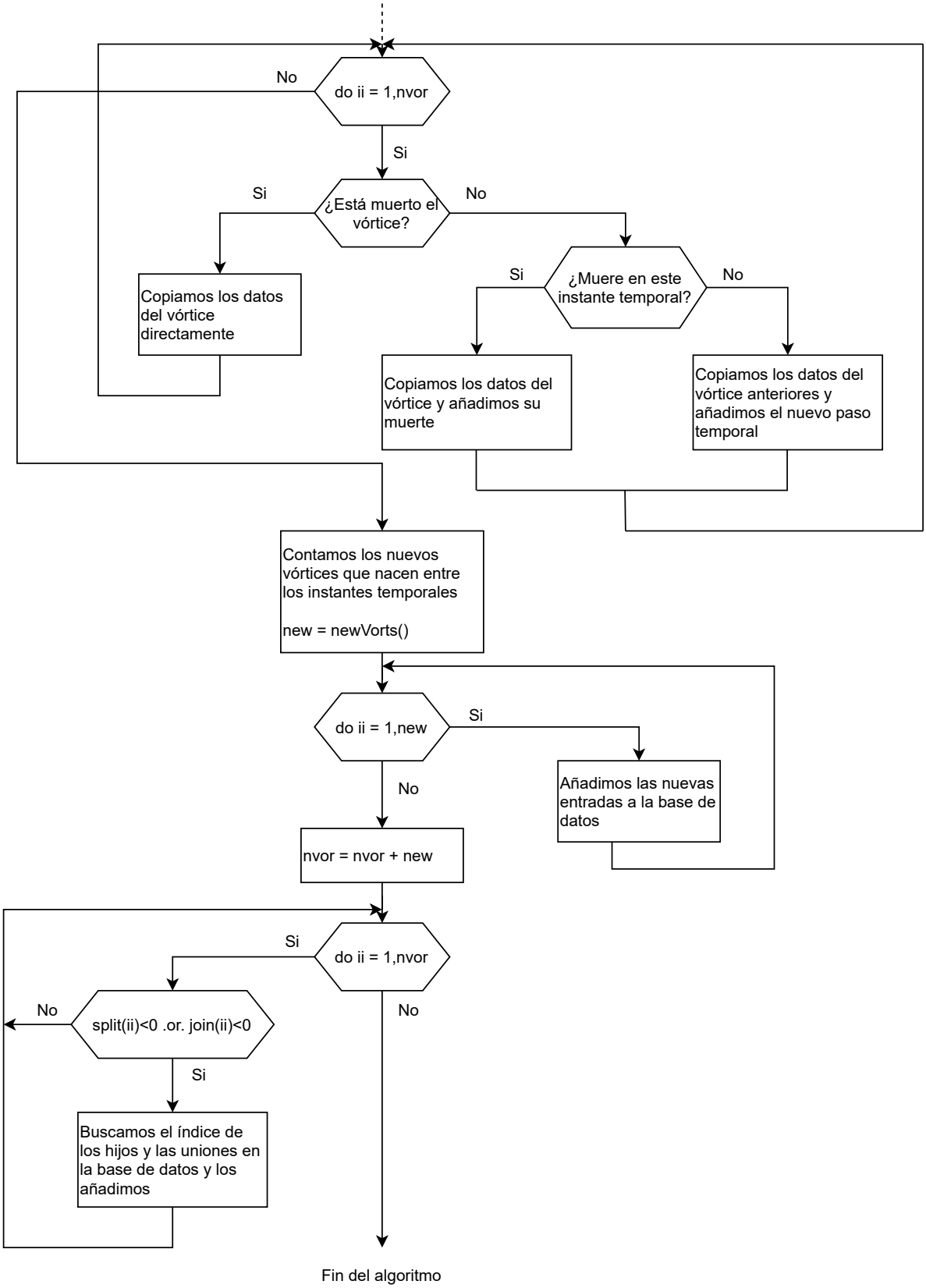


Figura 5.7: Diagrama de flujo simplificado de la actualización de la base de datos

5.4. Calibración y resultados

Una vez tenemos el código desarrollado y funcionando, antes de empezar a sacar resultados, tenemos que pasar por una fase de calibración de los parámetros que hemos introducido. De todos los parámetros que hemos introducido, únicamente el parámetro de las conexiones fáciles se puede calibrar al uso. El resto de parámetros son resultado del criterio que nosotros impongamos. Por ejemplo, el parámetro `RTHR` visto en las conexiones difíciles que sirve para dar prioridad al volumen de los vórtices o a las repeticiones, es elegido según nuestro criterio de identificación de vórtices. En el caso de analizar los resultados y ver que el criterio no es el adecuado se cambiaría de forma fácil mediante los parámetros. Lo mismo ocurre con el resto de parámetros vistos en el último filtro, por lo que nos vamos a centrar en el parámetro `DTHR`.

Para poder calibrar el parámetro es necesario tener una base de calibración, es decir, una solución correcta con la que podamos comparar. Esta solución correcta va a ser la obtenida identificando únicamente por conexiones difíciles y último filtro. Así, lo que se va a realizar es por un lado una identificación por conexiones fáciles y por otro lado una identificación por conexiones difíciles y último filtro, comparando las conexiones directas obtenidas y viendo cual es el error cometido. El valor de error que se va a imponer es que sea menor a 10^{-3}, es decir, un error de un 0.1 %.

$$p_e \leq 10^{-3} \tag{5.5}$$

Este valor se ha escogido porque hay que tener un compromiso entre muy poco error y velocidad. Si el error que buscamos es muy bajo, el umbral va a hacer que identifiquemos muy pocos vórtices por conexiones fáciles y el resto se tengan que identificar por conexiones difíciles, más costosas computacionalmente. Los resultados obtenidos tras haber usado 20 instantes temporales y haciendo la media de todos ellos son los siguientes:

Identificación por Chong		
`DTHR`	**Error** [%]	**Identificación** [%]
0.020	0.725	84.387
0.018	0.599	84.023
0.015	0.452	83.306
0.013	0.356	82.705
0.010	0.227	81.412
0.008	0.159	80.052
0.005	0.069	75.787

Identificación por Qs		
`DTHR`	**Error** [%]	**Identificación** [%]
0.020	0.167	91.072
0.018	0.159	90.955
0.015	0.134	90.729
0.013	0.119	90.522
0.010	0.105	90.104
0.008	0.080	89.580
0.005	0.044	87.885

Tabla 5.1: Calibración del parámetro `DTHR` de las conexiones fáciles para el criterio de Chong y para el criterio de Qs

Los resultados mostrados en la tabla 5.1 corresponden para un $Re_\tau = 500$. Como podemos ver, prácticamente con todos los valores del parámetro `DTHR` se obtiene un gran porcentaje

de identificación, esto quiere decir que la mayor parte de las identificaciones del programa se hace en las conexiones fáciles. Sin embargo, existen diferencias entre la identificación por Qs y la identificación por Chong. Como se vio en el apartado de resultados §4.3 la identificación por Qs estaba formada por grandes estructuras y esto facilita la identificación debido a que estas cambian poco de un instante temporal al siguiente, lo que se corresponde con un gran porcentaje de identificación, mientras que la identificación por Chong estaba formada por lo que se llaman *vortex clusters*, que son pequeños filamentos que se unen entre sí para formar vórtices más grandes. Estos filamentos son más fácil que se separen del cluster, por lo que la identificación se complica, además de que existe un mayor número de vórtices por instante temporal. Si atendemos al criterio de error que hemos establecido, el valor umbral para el caso de la identificación por Qs sería `DTHR` $= 0.01$ mientras que para la identificación por Chong sería `DTHR` $= 0.005$, siendo el umbral de Chong menor al de los eventos *uv* por los motivos que acabamos de contar.

Otro aspecto importante a destacar es que el porcentaje de identificación de conexiones fáciles baja mucho más rápido que lo hace el error. Al bajar el umbral, reducimos el número de conexiones erróneas pero también hay un mayor número de conexiones correctas que no son capaces de pasar el umbral y no las tomamos. Desde el punto de vista práctico no hay ningún problema, puesto que estas conexiones serán cogidas posteriormente en las conexiones difíciles y en el último filtro, pero a costa de aumentar el tiempo de cálculo. Es por esto que no se ha impuesto un criterio de error aún mayor, para no penalizar el tiempo del algoritmo.

A continuación vamos a mostrar que porcentaje de la identificación se hace en cada filtro, tanto por conexiones directas (vórtices que continúan), como conexiones indirectas (hijos y uniones).

	Identificación por Chong	
	Directas [%]	**Indirectas** [%]
Fáciles	87.25	0.00
Difíciles	11.92	90.57
Ult. Filtro	0.84	9.43
Total	100.00	100.00

	Identificación por Qs	
	Directas [%]	**Indirectas** [%]
Fáciles	97.05	0,00
Difíciles	2.77	95.14
Ult. Filtro	0.17	4.86
Total	100.00	100.00

Tabla 5.2: Porcentaje de identificación realiza por cada filtro del seguimiento temporal

En la tabla 5.2 podemos ver resultados interesantes. En primer lugar, la mayor parte de las conexiones directas se hace en el filtro de conexiones fáciles. Esto es algo que esperamos por la propia naturaleza del filtro, y el resto de conexiones directas que no han superado el umbral son cogidas después en los posteriores filtros. Debido a que en la identificación por Chong el umbral es más bajo que en la identificación por Qs, el porcentaje de conexiones fáciles que se obtienen es menor, y estas tienen que ser identificadas sobre todo el las conexiones difíciles, mientras que en la identificación por Qs casi todas las conexiones directas se hacen en las fáciles.

Otro aspecto importante a destacar es el porcentaje de identificación del último filtro. En

ambos casos la identificación de conexiones directas es despreciable y consigue identificar algunas conexiones indirectas, aunque no en gran medida. Es por esto que, debido al coste computacional que puede tener, podemos decidir no incluirlo en el programa y los resultados no van a variar significativamente.

Con respecto al análisis de las conexiones que se producen para cada tipo de identificación, tenemos la siguiente tabla que nos indica que porcentaje de conexiones directas, indirectas y no conexiones con respecto al total de vórtices que se analizan:

	Directas [%]	**Indirectas** [%]	**Sin Conexión** [%]
Chong	86.83	7.08	6.09
Qs	92.75	3.58	3.67

Tabla 5.3: Porcentaje de las distintas conexiones que existen para ambos tipos de identificación

El primer aspecto que podemos destacar de la tabla 5.3 es que la hipótesis que se hizo al inicio del capítulo de que ambos instantes temporales cambian poco es cierta. En ambos casos la mayor parte de los vórtices se identifican mediante conexión directa ya que el 90 % de los vórtices continúan en ambos casos. En el caso de identificación por Chong este porcentaje es menor debido a que sus estructuras son más pequeñas y son más susceptibles a romperse, por lo que aparecen mayor número de conexiones indirectas. En ambos casos, los vórtices que no tienen conexión se encuentran en torno al 5 %, que pueden ser por apariciones espontáneas, o por vórtices que no han podido ser identificados correctamente y se asignan como vórtices que mueren o nacen.

Vamos a analizar que tal se ha realizado la paralelización por memoria compartida por OpenMP, analizando lo que ocurre con los tiempos de ejecución cuando aumentamos el número de procesadores, tanto para la identificación de Qs como para Chong, para un $Re_\tau = 500$. Los resultados se muestran en la tabla 5.4 y en la figura 5.8.

Tiempo identificación por Chong [s]						
Procesadores OMP	1	2	4	6	8	10
Fáciles	13.840	7.025	3.637	3.603	3.640	3.640
Difíciles	9.838	5.225	2.807	1.937	1.540	1.310
Ult. Filtro	4.651	2.425	1.328	0.880	0.715	0.630
Total	28.328	14.675	7.772	6.420	5.894	5.581

Tiempo identificación por Qs [s]						
Procesadores OMP	1	2	4	6	8	10
Fáciles	2.748	1.388	0.707	0.528	0.469	0.488
Difíciles	4.889	2.988	1.535	1.096	0.892	0.745
Ult. Filtro	5.315	3.306	1.870	1.385	1.045	0.880
Total	12.952	7.682	4.112	3.008	2.407	2.112

Tabla 5.4: Tabla de tiempos al variar el número de procesadores en OpenMP para el seguimiento temporal a $Re_\tau = 500$

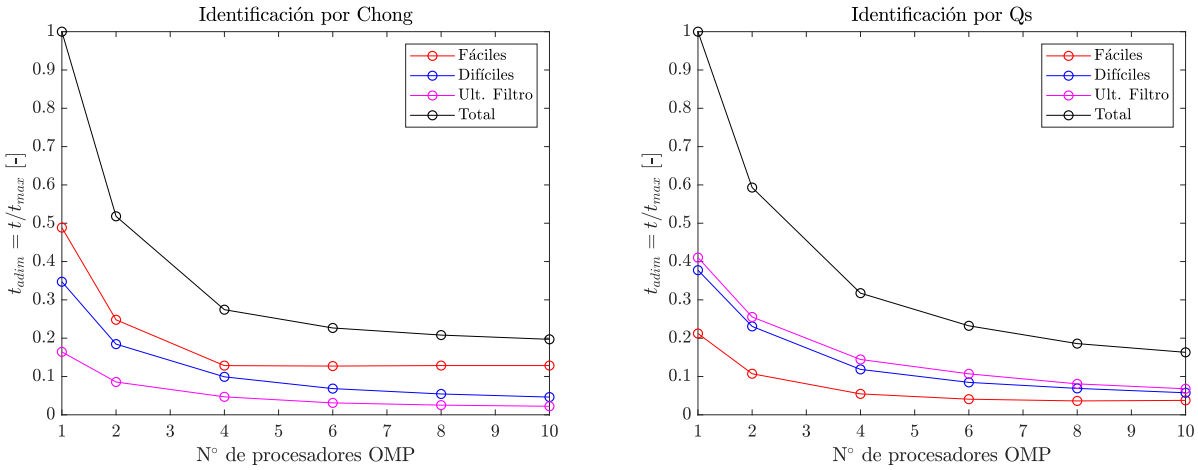


Figura 5.8: Visualización de la reducción del tiempo adimensional en el seguimiento temporal al aumentar el número de procesadores de OpenMP. A la izquierda, identificación por Chong. A la derecha, identificación por Qs

La figura 5.8 está adimensionalizada con el tiempo total máximo, que se produce con 1 procesador. Como podemos ver, al igual que ocurría con la paralelización MPI, la mayor reducción de tiempo ocurre cuando pasamos de ningún procesador a tener 2 o 4. Cuando aumentamos el número de procesadores existe mejora pero no es tan pronunciada. Esto nos dice que lo más rentable es usar un número de procesadores entre 2 y 4 para OpenMP. Además, debido a la configuración que tiene el superordenador de MareNostrum, que tiene 48 cores por nodo, este es el número máximo de procesadores que se pueden poner en memoria distribuida, pero lo normal es usar 12 procesadores para MPI y 4 para OpenMP o 24 para MPI y 2 para OpenMP, por lo que el algoritmo es apto para implementarse en MareNostrum.

Si atendemos al tiempo que tarda cada tarea, en el caso de identificación por Qs, tenemos pocos vórtices formados por un gran número de puntos por lo general. Al contar con menos vórtices tiene que hacer menos comprobaciones, por lo que los tiempos son más bajos que en el caso de Chon. Además, podemos ver que las conexiones difíciles y el último filtro son más lentos que las conexiones fáciles, y esto es debido a que estos filtros utilizan comparación punto a punto de los vórtices, lo cual es muy lento.

Por otro lado, si atendemos a la identificación por Chong, tenemos un mayor número de vórtices que en el caso de Qs, por lo que en general la identificación es mucho más lenta (tenemos que lidiar con más vórtices, ya sean más grandes o más pequeños). Pero el hecho de que tengan menos puntos hace que el tiempo de las conexiones difíciles y el último filtro sea menor, y esto es debido a que el número de vórtices que nos quedan después de las conexiones fáciles es pequeño y además tienen pocos puntos, por lo que no tarda excesivamente.

Finalmente, vamos a mostrar la evolución de un *ejection* desde su nacimiento hasta su muerte.

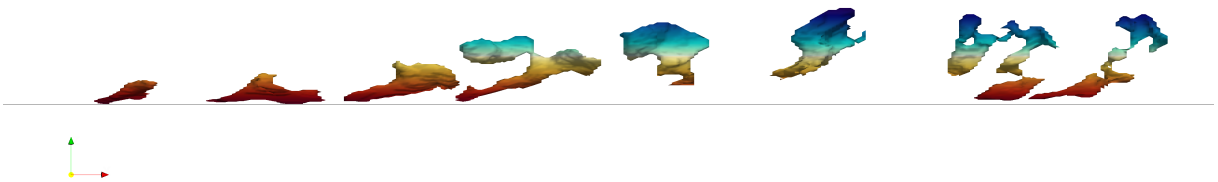


Figura 5.9: Ejemplo de la evolución de un *ejection* desde su nacimiento hasta su muerte para $Re_\tau = 500$

Como se puede apreciar en la figura 5.9, el *ejection* nace siendo muy pequeño cerca de la pared. Llega un momento en el que evoluciona y crece separándose de la pared, en el proceso que se conoce como *bursting*. Al final de la vida podemos ver como se estreha en una serie de filamentos y acaba rompiéndose en varios vórtices, finalizando así su vida.

El vórtice mostrado en la figura 5.9 es solo un ejemplo de la información que el algoritmo nos puede proporcionar, y con un tiempo de cálculo suficiente y un buen post-procesado de los datos podemos obtener estadísticas relativas al flujo para ver como influyen estos vórtices en el flujo en función del método de identificación y en función del número de Reynolds.

Capítulo 6

Integración con el código DNS

Este capítulo va a tratar de la integración de los algoritmos de identificación y seguimiento junto con el código DNS llamado LISO. El objetivo es que el código final sea capaz de calcular las estructuras coherentes al mismo tiempo que se van generando los campos, lo que recibe del nombre de cálculo *On The Fly*. La ventaja principal es que si solo deseamos tener información sobre el seguimiento temporal no necesitamos guardar en memoria el resultado de la identificación.

El código LISO nació en el año 1992 y hasta el día de hoy ha pasado por varias manos y ha sido modificado en numerosas ocasiones, siendo un código largo y difícil de entender. Nuestro objetivo es modificarlo de tal manera que podamos integrar nuestro código sin penalizar la velocidad del código DNS.

Como se vio en la sección §3.5 donde se explicaban las características del superordenador MareNostrum, este consta de nodos de procesadores con una memoria RAM muy alta, de hasta 384GB por nodo. Esta característica va a ser utilizada para crear una memoria virtual dentro del propio programa, evitando la escritura a disco y aumentando así la velocidad total del código.

A continuación se explicará la solución empleada y se detallará el manejo de la memoria para que el código DNS y nuestro algoritmo funcionen de forma eficiente.

6.1. Solución empleada

La solución planteada es el programar diversos grupos de paralelización MPI, y enfocar cada grupo a una parte del código en concreto. Esto nos permite que el código sea modular, ya que cada tarea tendrá su propio grupo de procesadores, con su procesador maestro y el resto de esclavos. Se crearán intercomunicadores entre los grupos que nos va a permitir comunicarlos, así cada uno funcionará independientemente hasta que llegue el momento

de la comunicación.

En total se van a crear 3 grupos MPI, cada uno enfocado en las siguientes partes del código:

- **Código DNS:** el primer grupo es el que se encarga de realizar el cálculo DNS que existía antes de introducir nuestro módulo de post-proceso. Realiza las mismas tareas que hacía anteriormente pero ahora debe enviar a la memoria virtual el campo booleano con el que se hará la identificación de vórtices.
- **Memoria virtual:** este grupo es el que se encuentra entre el código DNS y el código de post-proceso, y se encarga de recibir y enviar datos entre ellos. Además, se encarga de almacenar en RAM todos los datos necesarios evitando la escritura a disco, lo que aumenta la velocidad del código.
- **Post-procesado de vórtices:** este último grupo es el que se encarga de realizar las tareas de identificación y seguimiento temporal de los vórtices, ya explicada en los capítulos anteriores.

La comunicación entre distintos grupos de procesadores tiene la siguiente estructura:

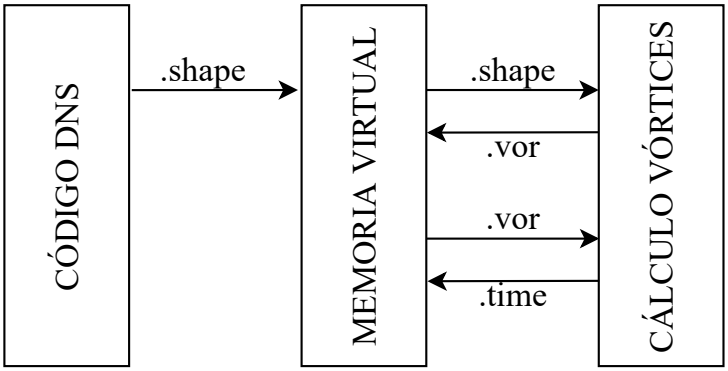


Figura 6.1: Diagrama esquemático de la estructura que siguen los envíos entre los grupos de procesadores

En la figura 6.1 nos referimos como "shape", ".vor" y ".time" a la extensión de los archivos que guarda cada tipo de dato. Los archivos ".shape" son los que contienen la matriz booleana para la identificación, los archivos ".vor" contienen los resultados de la identificación de vórtices y los archivos ".time" tienen la base de datos resultante del seguimiento temporal. Aunque se ha usado esta nomenclatura, los archivos no se escriben a disco, se mantienen en memoria RAM.

El aspecto más importante a destacar es que el código DNS únicamente envía datos a la memoria virtual y no recibe ninguno. Esto hace que cuando antes se guardaba a disco, ahora se realice una comunicación MPI, por lo que el código no se ve ralentizado.

La memoria virtual se encuentra siempre preparada para recibir datos de los otros dos grupos, evitando ser un cuello de botella del programa.

El código LISO tiene un pequeño archivo de texto del que lee las características del cálculo, como por el ejemplo el número de Reynolds al que queremos lanzar la simulación o el número de pasos temporales que queremos calcular. En este archivo vamos a añadir una sección en la que especificaremos el número de procesadores que va a ir encargado a cada tarea. Cuando lanzamos el programa, el número de procesadores MPI especificados debe ser igual a la suma de los procesadores individuales para cada grupo especificados en el archivo, en caso contrario dará error y no se ejecutará.

Tras iniciarse MPI, el función del ID global que reciba cada procesador lo asignamos a un grupo de procesadores. Una vez tenemos los grupos hechos, generamos los intercomunicadores que se encargarán de comunicar un grupo con otro. En este caso no hará falta crear el intercomunicador entre el código DNS y el cálculo de vórtices puesto que no van a enviarse datos directamente.

Antes de incorporar la funcionalidad de los grupos MPI el código usaba el comunicador `MPI_COMM_WORLD` para realizar todas las comunicaciones. Ahora hay que usar el comunicador propio de cada grupo para estas funciones, por lo que el código ha tenido que ser adaptado para este cambio.

Finalmente tenemos el código DNS preparado para funcionar y solo falta añadir el envío de la matriz booleana con el que empieza el proceso de identificación. Un aspecto inherente a trabajar en problemas de turbulencia y que son aspectos que en otro tipo de problemas no se tiene en cuenta es el tamaño de las matrices. Para hacer un envío por MPI, tienes que especificar el número de elementos que vas a mandar en el mensaje y este número tiene un límite que viene marcado por un entero con signo en Fortran. El límite para un entero de 4 bytes es de aproximadamente 2147 millones de elementos, mientras que si tomamos el caso de $Re_\tau = 1000$ mostrado en la tabla 4.1 tenemos que el número de puntos es de $N_x N_y N_z = 2710$ millones de puntos aproximadamente, lo que hace que no podamos mandarlo en un único envío.

Existen dos posibles soluciones para este problema: podemos dividir la matriz en varios envíos, pero tiene el problema que tiene que establecer comunicación tantas veces como envíos existan, y esto puede ser especialmente lento en un superordenador si los procesadores que comunican se encuentran alejados entre sí; y otra solución es cambiar el tipo de dato que estamos mandando. Si nuestra matriz es de tipo entero de 4 bytes, si especificamos este tipo de elemento en el envío, MPI va a ir mandando paquetes de 4 bytes con nuestros datos, pero si especificamos otro tipo de dato que tenga un tamaño de paquete más grande, como complejo de precisión doble que tiene 32 bytes, estaremos enviando 8 veces más información por paquete y el número de elementos a enviar será menor. En este caso se ha implementado dividir la matriz en varios envíos porque esta segunda solución puede llegar a no ser suficiente cuando aumentamos demasiado el número de Reynolds, mientras que dividiendo el envío podemos hacer matrices lo suficientemente pequeñas para poder ser enviadas en un único mensaje.

Procedemos a explicar la estructura de la memoria virtual y del post-procesado de vórtices.

6.2. Memoria virtual

La memoria virtual tiene como objetivo ser un intermediario entre el código DNS y el post-proceso de vórtices pasando y recibiendo información cuando los otros dos grupos de procesadores lo requieran para evitar ralentización en el código. Además, se encarga de guardar en memoria RAM los datos necesarios, evitando la escritura a disco.

Esta memoria tiene una estructura de procesadores MPI que consta de 2 procesadores maestros: un procesador se encarga de la comunicación con el código DNS y otro procesador se encarga de la comunicación con el código de post-proceso, y trabajan por separado. El resto de procesadores se dedican a recibir y enviar datos según se lo vayan solicitando los otros dos procesadores maestros.

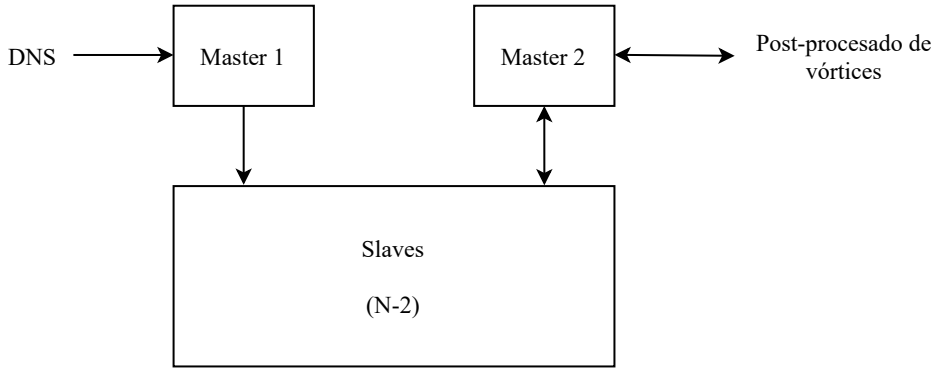


Figura 6.2: Estructura de los procesadores MPI de la memoria virtual

El proceso que realiza cada procesador es el siguiente:

- **Maestro 1:** este es el procesador que se encarga de la comunicación con el código DNS. El procesador esta todo el rato esperando a recibir del código LISO para evitar que este tenga que esperar a la memoria virtual, evitando cualquier tipo de ralentización. Una vez tenemos la matriz booleana la mandamos al procesador esclavo que corresponda.
- **Maestro 2:** este procesador se encarga de la comunicación con el código de post-proceso. Lo primero que hace es interrogar al procesador esclavo si tiene una matriz booleana, si no la tiene espera un segundo y vuelve a interrogar, si la tiene, la recibe y la manda para realizar la identificación y espera a recibir los resultados de la identificación. Una vez recibe los resultados los manda de vuelta al procesador esclavo correspondiente y se repite este ciclo. Cuando tenemos un número determinado de pasos temporales identificados (según un parámetro del cálculo), el procesador pide

a los esclavos que envíen los datos de la identificación por orden y los manda al seguimiento temporal, el cual devuelve los vórtices muertos una vez haya acabado con este seguimiento y los manda al procesador esclavo correspondiente. Cuando hemos llegado al número máximo de pasos temporales y ya hemos identificado todos los pasos temporales, el maestro envía los últimos pasos temporales que no se les ha hecho el seguimiento y finalmente manda al post-procesado de vórtice todos los vórtices muertos que hemos ido almacenando para que se reconstruya la base de datos y se guarde en disco finalmente.

- **Esclavos:** son el resto de procesadores, en total N-2 donde N es el número de procesadores del grupo. La comunicación entre maestros-esclavos se hace a través de *flags*. Todos los procesadores están esperando a recibir de cualquier procesador una *flag*. Cuando reciben un mensaje, en función del valor de esta *flag* ya saben si tienen que recibir o enviar algún dato y de quien. Cuando envían algún dato, por ejemplo cuando envían la matriz booleana para su identificación, o cuando mandan los resultados de la identificación para hacer su seguimiento temporal, se borran de la memoria, por lo que si queremos conservar algún dato debe haber sido guardado anteriormente.

Vamos a poner un ejemplo de trabajo del maestro 2 que es el más complicado de entender. Supongamos que vamos a resolver 23 pasos temporales y vamos a hacer el seguimiento temporal cada 5 pasos temporales. El procesador va a ir mandando hacer identificaciones hasta que tenga 5 pasos temporales y cuando los consigue recoge todos los ".vor" que ha mandado a los esclavos y los manda al post-procesado de vórtices, el cual le devuelve los vórtices muertos después del seguimiento. Este proceso se repite cuando llegamos a 10, 15 y 20 pasos temporales. Cuando terminamos los 3 pasos temporales el procesador tiene que mandar a hacer el seguimiento de estos últimos 3 pasos temporales y en este caso no pide que le devuelvan los vórtices muertos, ya que a continuación va a mandar todos los vórtices muertos que teníamos guardados en memoria para que el post-procesado de vórtices se encargue de montar la base de datos al completo y la guarde en memoria.

Otro aspecto importante es saber cómo se guarda la memoria en los procesadores esclavos. En la configuración que se ha implementado todos los procesadores guardan todo tipo de datos, es decir, no hay procesadores que se dediquen a guardar un único tipo de dato. Cuando los procesadores maestros mandan guardar datos a los esclavos los mandan en orden, por ejemplo, el maestro 1 mandará el primer campo al esclavo 1, el segundo al esclavo 2, y así sucesivamente hasta llegar al esclavo N-2 que después volvería al esclavo 1 otra vez. En el caso del maestro 2 que envía y recibe datos de los esclavos tendrá que tener dos contadores para cada tipo de dato, es decir, si los últimos datos de la identificación los ha mandado al esclavo 5 por ejemplo, cuando vaya a recibir los datos para el seguimiento tendrá que empezar por el procesador 1 hasta llegar al 5. El maestro 2 seguirá guardando datos de la identificación del esclavo 6 en adelante, y cuando tenga que volver a recibir datos en este caso empezará recibiendo del esclavo 6.

De esta forma nos aseguramos que todos los procesadores tienen aproximadamente la misma memoria almacenada en RAM no tenemos ningún procesador saturado.

6.3. Post-procesado de vórtices

Finalmente tenemos el post-procesado de vórtices, que se encarga de preparar todos los procesadores antes de lanzar los algoritmos de identificación y de seguimiento temporal. En este caso, la estructura de los procesadores es mucho más sencilla que la que nos podemos encontrar en la memoria virtual por ejemplo, ya que tenemos una configuración sencilla de maestro-esclavos.

La comunicación con la memoria virtual se hace a través de *flags*. Cuando el grupo de procesadores entra en la subrutina de post-proceso, el maestro se queda esperando recibir una *flag* de la memoria virtual, la cual es transmitida a todos los procesadores del grupo para realizar una tarea. En función del valor de esta variable la subrutina puede hacer tres tareas:

- Si *flag* = 1 nos indica que la tarea a realizar es la identificación de vórtices, por lo que el procesador maestro se dispondrá a recibir la matriz booleana y ya estará todo listo para la identificación.
- Si *flag* = 2 nos indica que procedemos a realizar el seguimiento temporal de los distintos pasos temporales. El procesador maestro va recibiendo los resultados de la identificación y los va distribuyendo entre los procesadores que se van a encargar de calcular los vectores conexión. Una vez que ha terminado de repartir los datos espera a que estos terminen y se dedica a actualizar la estructura. Cuando ha terminado de actualizar la estructura elimina los vórtices muertos de la base de datos y los manda de vuelta a la memoria virtual.
- Si *flag* = 3 nos indica que hemos acabado con la identificación y vamos a generar la base de datos completa, por lo que va a recibir todos los vórtices muertos de la memoria virtual y va a ir actualizando la base de datos. Finalmente ordena los vórtices según su índice y finalmente escribe a disco los resultados.

Los algoritmos de identificado y seguimiento temporal son los ya vistos en capítulos anteriores, por lo que no requieren de una mayor explicación.

Capítulo 7

Conclusiones y trabajos a futuro

7.1. Conclusiones

Tras todo lo realizado a lo largo de este documento y viendo los objetivos planteados, podemos sacar una serie de conclusiones de este trabajo:

- La potencia computacional necesaria para resolver problemas de turbulencia escala de forma exponencial con el número de Reynolds, siendo necesario aplicar herramientas de paralelización de nuestros algoritmos para poder ser implementados en supercomputadores como MareNostrum.
- El primer algoritmo desarrollado es el de identificación de vórtices, el cual usa un punto origen de cada vórtice y lo recorre en las 6 direcciones cartesianas hasta encontrar todos los puntos que pertenecen al vórtice. Una vez identificados, se obtienen las características de los vórtices con los que poder hacer estadísticas, como volumen, centro de masas, caja envolvente y ubicación dentro del dominio. Este algoritmo ha sido validado mediante una matriz booleana de calibración y lo ha resuelto correctamente, por lo que su desarrollo ha sido satisfactorio.
- El segundo algoritmo se encarga de encontrar un mismo vórtice en distintos instantes temporales y hacer su seguimiento, identificando el nacimiento, muerte, evolución, rotura o coalescencia de los vórtices. Para ello se han diseñado tres filtros, uno que se encarga de identificar los vórtices muy similares entre instantes temporales y los otros dos hacen una identificación mas avanzada buscando hijos y uniones de los vórtices. Este algoritmo consta de parámetros que han sido calibrados y se ha observado un correcto seguimiento de los vórtices, por lo que ha sido desarrollado correctamente.
- Ambos algoritmos se han diseñado para ser ejecutados mediante paralelización de memoria distribuida (MPI) y de memoria compartida (OpenMP). Esto permite

abordar problemas más grandes haciendo uso de todos los recursos computacionales que tenemos a nuestro alcance.

- La integración de los algoritmos con el código DNS nos permite calcular simultáneamente el campo fluido junto a los vórtices. Esta integración se ha diseñado para hacer uso de los módulos de alta memoria RAM de MareNostrum, evitando así la escritura a disco y esto aumenta la velocidad del código.
- Se ha diseñado una memoria virtual, la cual se dedica a recibir, enviar y almacenar los datos entre el código DNS y el código de post-proceso de vórtices. Esta memoria evita que exista ralentizamiento de la DNS, el cual era un objetivo fundamental de la integración.

7.2. Trabajos a futuro

A lo largo de todo este trabajo se ha desarrollado un algoritmo que es capaz de identificar y seguir temporalmente estructuras coherentes, lo que nos permite obtener una base de datos de la que sacar estadísticas y estudiar la física que hay detrás de los vórtices. Puesto que este trabajo ha estado centrado únicamente en la parte computacional del problema, podemos establecer unas pautas a seguir para continuar este trabajo:

- Generar programas de post-proceso que permitan analizar la base de datos de vórtices, tanto del punto de vista estático como dinámico, obteniendo así resultados útiles en el estudio de la física de las estructuras coherentes.
- Aumentar el número de Reynolds de las simulaciones. En este trabajo se han empleado simulaciones hasta un número de Reynolds máximo de $Re_\tau = 1000$ y la bibliografía muestra resultados hasta un $Re_\tau = 4200$ en [23]. Este valor puede ser aumentado gracias a la paralelización de los algoritmos y a la integración junto al código DNS, lo que permite ahorrar la escritura de campos en disco.
- Implementar nuevos criterios de identificación mostrando las diferencias y similitudes con los dos empleados en este trabajo.

Desde el punto de vista de mejora del código, se pueden realizar una serie de ampliaciones:

- La unificación de estructuras vista en la identificación de vórtices la realiza únicamente un procesador. Esto puede llegar a ser un cuello de botella en el código, por lo que habría que idear una forma de paralelizarlo.
- La actualización de la base de datos en el algoritmo temporal también la hace un único procesador, por lo que sería interesante pensar una forma de paralelizarla mediante memoria compartida.

- En el caso de tener dos vórtices próximos que se tocan por una ramificación, el código no es capaz de distinguirlos y lo identifica como un único vórtice. Esto puede ser solucionado encontrando los núcleos de los vórtices y utilizarlos para realizar la identificación.
- Los filtros usados en el seguimiento temporal pueden ser sustituidos por el uso de un algoritmo de *machine learning*, el cual pueda estimar un campo futuro a partir del estado actual, comparar ambos campos y realizar así la identificación.

Capítulo 8

Pliego de condiciones y presupuesto

8.1. Pliego de condiciones

Cualquier trabajador a la hora de realizar su trabajo esta sometido a una serie de riesgos que pueden afectar tanto a su salud como al rendimiento del mismo. Es por esto que la Ley 31/1995 del 8 de noviembre, de Riesgos Laborales, se encarga de determinar el nivel mínimo de garantías para proporcionar al trabajar un nivel de protección adecuado frente a los riesgos de las condiciones de trabajo.

En nuestro caso en concreto nos centraremos en los riesgos asociados al uso de pantallas de visualización (PVD), que vienen recogidos en el Real Decreto 488/1997 del 14 de abril, en la que se define como pantalla de visualización cualquier pantalla alfanumérica o gráfica. El equipo completo esta constituido por una pantalla de visualización, un teclado que nos permita la adquisición de datos, un programa para la interconexión persona/máquina, de accesorios ofimáticos y una silla y una mesa como superficie de trabajo. La evaluación de los riesgos se toman a través de las características del puesto, entre las que destacan:

- El tiempo promedio de utilización del equipo en un día.
- El tiempo máximo requerido para una atención continua a la pantalla por una tarea.
- El grado de atención que exija la tarea en cuestión.

Así, los riesgos a los que está sometido un trabajador en este tipo de actividades se pueden clasificar en:

- **Seguridad:** debido a posibles caidas o contactos eléctricos en el puesto de trabajo.
- **Higiene industrial:** relacionado con la iluminación, ruido y condiciones termo-higrométricas.

- **Ergonomía:** causados por la fatiga visual, física o mental.

Las disposiciones mínimas que deben tener el puesto de trabajo para alcanzar los objetivos del Real Decreto son los siguientes.

Equipo

- **Pantalla:** los caracteres de la pantalla deben estar bien definidos y visualizados de forma clara, con una dimensión suficiente y un espaciado suficiente entre caracteres y renglones. La imagen debe ser estable, pudiéndose ajustar fácilmente la luminosidad, orientación e inclinación a voluntad del trabajado.
- **Teclado:** debe permitir su inclinación para que el trabajador adopte una postura cómoda, además de existir espacio suficiente para poder apoyar las muñecas y los brazos. La superficie del teclado deberá ser mate para evitar reflejos y la disposición de teclas deberá ser tal que se tienda a facilitar su utilización.
- **Mesa o superficie de trabajo:** esta superficie debe ser poco reflectante y debe tener espacio suficiente para poder albergar la pantalla, el teclado y los documentos que necesitemos. El soporte de los documentos debe ser estable y regulable, diseñado para minimizar los movimientos incómodos de cabeza y de ojos.
- **Asiento de trabajo:** este deberá ser estable, ajustable tanto en altura como en reclinación y debe permitir la libertad de movimiento del trabajador, además de procurar una postura confortable.

Entorno

- **Espacio:** el puesto de trabajo deberá tener un espacio suficiente para permitir los cambios de postura y los movimientos en el trabajo.
- **Iluminación:** la iluminación debe ser suficiente para realizar las tareas necesarias del trabajo, además de evitar reflejos y deslumbramientos molestos en la pantalla o en otras partes del equipo.
- **Ruido:** el ruido que producen los distintos equipos debe ser tenido en cuenta para evitar que perturben la atención ni la palabra.
- **Calor y humedad:** deberán mantenerse en niveles aceptables para evitar molestias en los trabajadores.

El entorno en el que se ha desarrollado este trabajo cumple con todo lo especificado en el Real Decreto 488/1997 del 14 de abril.

8.1.1. Recursos informáticos

Este trabajo ha necesitado de una serie de recursos informáticos para ser realizado, ya que al no contar con parte experimental todas las simulaciones han tenido que ser resueltas mediante ordenador. Podemos diferenciar estos recursos tanto en hardware como en software.

Hardware

La mayor parte de este trabajo se ha desarrollado en 3 máquinas, las cuales tienen las siguientes especificaciones:

- **Estación de trabajo:** donde se han realizado los algoritmos y las pruebas para su correcto funcionamiento para números de Reynolds bajos. Está formado por 2xAMD Epyc 7281, lo que da en total 32 cores y 64 hilos de procesamiento, 128GB DDR4 de memoria RAM, 512GB de disco duro SSD y 1TB de disco duro HDD convencional.
- **MareNostrum:** supercomputador perteneciente al *Barcelona Supercomputing Center* con el que se han realizado las simulaciones más exigentes de nuestros algoritmos. Sus especificaciones técnicas ya han sido descritas en la sección §3.5.
- **Portátil:** usado para conexión remota a los dos equipos, documentación bibliográfica y redacción de este documento. Es el modelo Lenovo G500s, que consta de un procesador Intel i7-3632QM de 8 procesadores 8 hilos, 16GB DDR3 de memoria RAM, 1TB de disco duro SSD y 512GB de disco duro HDD convencional.

Software

El trabajo ha sido desarrollado en el sistema operativo GNU-Linux, haciendo uso de programas de software libre cuando ha sido posible. El software necesario para ejecutar estos algoritmos es el siguiente:

- `ifort` (comercial) es el compilador del código que ha sido desarrollado en Fortran. Este ya incluye las librerías necesarias para la paralelización por memoria compartida OpenMP. El precio de una licencia a nivel profesional es de 1500$, lo que al cambio son 1267€ a día de redactar esta memoria.
- `HDF5` (libre) estándar que nos permite la escritura y lectura de datos a disco de forma eficiente y de forma paralela.
- `FFTW3` (libre) nos permite hacer operaciones en transformada de Fourier de forma muy eficiente, usada en el código DNS.

- `MPI` (libre) estándar que permite la paralelización por memoria distribuida usada en nuestros algoritmos y en el código DNS.
- `VScode` (gratuito) programa que proporciona una interfaz para que la programación sea mas sencilla.
- `MATLAB` (comercial) usado como programa de post-proceso y para obtener diversos resultados y gráficas. El coste de la licencia educacional permanente es de 500$, unos 422€ a día de redactar esta memoria.
- `paraview` (libre) herramienta open-source que nos permite obtener visualizaciones renderizadas de alta calidad.

8.2. Presupuesto

El presupuesto es el cálculo de los gastos asociados a la realización de un proyecto. En este caso se van a tener en cuenta por un lado el coste del número de horas que se han dedicado a la realización del proyecto, y por otro lado el coste de los materiales y de las licencias usadas a lo largo del trabajo.

8.2.1. Dedicación horaria a cada actividad

Podemos desglosar el número de horas dedicadas en función de las distintas tareas que se han tenido que realizar a lo largo de este proyecto. El listado de tareas es el siguiente:

1. **Familiarización con las herramientas existentes:** las primeras tareas del proyecto consistieron en aprender el uso de herramientas como el propio lenguaje Fortran, OpenMP y MPI, entender como funcionaba el código LISO y revisar la bibliografía actual sobre el estudio de la turbulencia a partir de estructuras coherentes.
2. **Algoritmo de identificación de vórtices:** se incluye tanto el desarrollo conceptual del algoritmo implementado en serie hasta la final implementación mediante paralelización de memoria distribuida.
3. **Algoritmo de seguimiento temporal de vórtices:** incluye el desarrollo conceptual, la implementación mediante una paralelización híbrida y además la calibración del mismo en función del método de identificación utilizado.
4. **Integración con código LISO:** desarrollo de la memoria virtual que evita la escritura a disco y la división de los procesadores MPI en distintos grupos, cada uno enfocado en una parte del código.
5. **Redacción:** finalmente en el apartado de redacción incluimos la redacción de la memoria y la generación de las figuras del proyecto.

El número de horas dedicada a cada tarea viene resumida en la tabla 8.1, en la que se ha supuesto que el coste económico por hora de un ingeniero es de 12€/h mientras que el coste de los tutores del proyecto, los cuales son doctores, es de 36€/h.

Tareas	**Concepto**	**Dedicación [h]**	**Tasa horaria [€/h]**	**Subotal [€]**
Tarea 1	Horas de Ingeniero	40	12	480.00
	Horas de Doctor	8	36	288.00
Tarea 2	Horas de Ingeniero	280	12	3360.00
	Horas de Doctor	20	36	720.00
Tarea 3	Horas de Ingeniero	300	12	3600.00
	Horas de Doctor	25	36	900.00
Tarea 4	Horas de Ingeniero	180	12	2160.00
	Horas de Doctor	20	36	720.00
Tarea 5	Horas de Ingeniero	100	12	1200.00
	Horas de Doctor	10	36	360.00
Total	Horas de Ingeniero	900	12	10800.00
	Horas de Doctor	83	36	2988.00
			Total	13788.00

Tabla 8.1: Costes asociados a las horas de trabajo empleadas en la realización de este proyecto.

8.2.2. Costes de equipos y software

Procedemos a hacer un desglose de los costes de los equipos y de las licencias de software. Para el coste de los equipos informáticos que se han usado se va a hacer una estimación en función del uso que se le ha dado y la vida media que tienen este tipo de productos. El tiempo de vida para ambos ordenadores es de unos 60 meses, por lo que si el coste de la estación de trabajo han sido 6000€, el coste es de unos 100€ al mes. El ordenador portátil ha tenido un coste de 650€, lo que da un coste de 10.86€ al mes. La duración de este proyecto ha sido de Febrero de 2020 hasta Agosto de 2020, lo que hace una duración total de 7 meses.

Categoria	**Concepto**	**Número [mes]**	**Coste [€/mes]**	**Subotal [€]**
Hardware	Estación de trabajo	7	100	700.00
	Ordenador portátil	7	10.86	76.02
			Total	776.02

Tabla 8.2: Costes asociados a las estaciones de trabajo

El coste del supercomputador MareNostrum se estima de otra forma. El precio se calcula mediante el número de horas por procesador que consumamos, y el coste del mismo es de 0.01€ por hora y por procesador. Para realizar las simulaciones de este trabajo se han usado en total unas 200000 horas, lo que dan los siguientes costes.

Categoria	Concepto	Número [h]	Coste [€/h]	Subotal [€]
Hardware	Supercomputador	200000	0.01	2000.00
			Total	2000.00

Tabla 8.3: Costes asociados al supercomputador

Finalmente, hay que sumar el coste de las licencias de software que es el siguiente:

Categoria	Concepto	Número	Coste [€]	Subotal [€]
Software	Intel Fortran	1	1267.00	1267.00
	Matlab	1	422.00	422.00
			Total	1689.00

Tabla 8.4: Costes asociados a las licencias del software

8.2.3. Costes totales del proyecto

Finalmente el coste total del proyecto es la suma de las horas de trabajo dedicadas, del hardware y del software, consideramos unos gastos indirectos de un 20 %, un 6 % de beneficio industrial y un 21 % de IVA en España. El desglose de los costes totales es el siguiente:

Categoria	Subotal [€]
Horas de trabajo	13788.00
Hardware y Software	4465.02
Gastos Indirectos	3650.60
Beneficio Industrial	1095.18
IVA	3833.13
Total	26831.93

Tabla 8.5: Coste total del proyecto incluyendo horas de trabajo, hardware y software, gastos indirectos, beneficio industrial e IVA

La estimación del coste total de este proyecto es **26831.93€**.

Bibliografía

[1] Osborne Reynolds. On the dynamical theory of incompressible viscous fluids and the determination of the criterion. *Philosophical transactions of the Royal Society of London.(a.)*, (186):123–164, 1895.

[2] Javier Jiménez. Near-wall turbulence. *Physics of Fluids*, 25(10):101302, 2013.

[3] Stephen B Pope. Turbulent flows, 2001.

[4] Ivan Marusic, Jason P Monty, Marcus Hultmark, and Alexander J Smits. On the logarithmic region in wall turbulence. *Journal of Fluid Mechanics*, 716, 2013.

[5] Joseph Boussinesq. *Essai sur la théorie des eaux courantes*. Impr. nationale, 1877.

[6] Joseph Smagorinsky. General circulation experiments with the primitive equations: I. the basic experiment. *Monthly weather review*, 91(3):99–164, 1963.

[7] Javier Jiménez. Computers and turbulence. *European Journal of Mechanics-B/Fluids*, 79:1–11, 2020.

[8] Robert Sugden Rogallo. *Numerical experiments in homogeneous turbulence*, volume 81315. National Aeronautics and Space Administration, 1981.

[9] John Kim, Parviz Moin, and Robert Moser. Turbulence statistics in fully developed channel flow at low reynolds number. *Journal of fluid mechanics*, 177:133–166, 1987.

[10] Sergio Hoyas, Martin Oberlack, Stefanie Kraheberger, and Francisco Alcantara-Avila. Turbulent channel flow at re τ= 10000. *APS*, pages H19–001, 2019.

[11] George Keith Batchelor and Albert Alan Townsend. The nature of turbulent motion at large wave-numbers. *Proceedings of the Royal Society of London. Series A. Mathematical and Physical Sciences*, 199(1057):238–255, 1949.

[12] Stephen J Kline, William C Reynolds, FA Schraub, and PW Runstadler. The structure of turbulent boundary layers. *Journal of Fluid Mechanics*, 30(4):741–773, 1967.

[13] Philip G Saffman. *Vortex dynamics*. Cambridge university press, 1992.

[14] Brenden Epps. Review of vortex identification methods. In *55th AIAA aerospace sciences meeting*, page 0989, 2017.

[15] Ronald J Adrian, Carl D Meinhart, and Christopher D Tomkins. Vortex organization in the outer region of the turbulent boundary layer. *Journal of fluid Mechanics*, 422:1–54, 2000.

[16] Juan C Del Álamo, Javier Jimenez, Paulo Zandonade, and Robert D Moser. Self-similar vortex clusters in the turbulent logarithmic region. *Journal of Fluid Mechanics*, 561:329, 2006.

[17] Sergio Hoyas and Javier Jiménez. Scaling of the velocity fluctuations in turbulent channels up to re τ= 2003. *Physics of fluids*, 18(1):011702, 2006.

[18] Sergio Hoyas and Javier Jiménez. Reynolds number effects on the reynolds-stress budgets in turbulent channels. *Physics of Fluids*, 20(10):101511, 2008.

[19] V Avsarkisov, S Hoyas, M Oberlack, and Jose Pedro Garcia-Galache. Turbulent plane couette flow at moderately high reynolds number. *Journal of Fluid Mechanics*, 751, 2014.

[20] Philippe R Spalart, Robert D Moser, and Michael M Rogers. Spectral methods for the navier-stokes equations with one infinite and two periodic directions. *Journal of Computational Physics*, 96(2):297–324, 1991.

[21] Sanjiva K Lele. Compact finite difference schemes with spectral-like resolution. *Journal of computational physics*, 103(1):16–42, 1992.

[22] Min S Chong, Anthony E Perry, and Brian J Cantwell. A general classification of three-dimensional flow fields. *Physics of Fluids A: Fluid Dynamics*, 2(5):765–777, 1990.

[23] Adrian Lozano-Duran and Javier Jimenez. Time-resolved evolution of coherent structures in turbulent channels. *APS*, pages D20–001, 2012.

[24] SS Lu and WW Willmarth. Measurements of the structure of the reynolds stress in a turbulent boundary layer. *Journal of Fluid Mechanics*, 60(3):481–511, 1973.

[25] M. Curcic. *Modern Fortran: Building efficient parallel applications*. Manning Publications, 2020.

[26] The hdf5 library & file format. `https://www.hdfgroup.org/solutions/hdf5/`.

[27] Hdf5 software documentation. `https://support.hdfgroup.org/HDF5/doc/index.html`.

[28] Mpi reduce and allreduce. `https://mpitutorial.com/tutorials/mpi-reduce-and-allreduce/`.

[29] Mpi documentation. `https://www.mpich.org/static/docs/latest/`.

[30] Roman Trobec, Bostjan Slivnik, Patricio Bulic, and Borut Robic. *Introduction to Parallel Computing*. Springer, 2018.

[31] Miguel Hermanns. Parallel programming in fortran 95 using openmp. *Technique Report, Universidad Politecnica De Madrid*, 2002.

[32] Top500, the list. `https://top500.org/`.

[33] Marenostrum4 user's guide. `https://www.bsc.es/user-support/mn4.php`.