UNIVERSITAT POLITÈCNICA DE VALÈNCIA

Escuela Técnica Superior de Ingeniería del Diseño

Universitat Politècnica de València

Escuela Técnica Superior de Ingeniería del Diseño

Trabajo Fin de Grado

Grado en Ingeniería Aeroespacial

Optimización de un código clásico de turbulencia

- Curso 2019/2020 -

Autor:

Eric González Caballero

Director:

Dr. Sergio Hoyas Calvo

Julio de 2020

A mis padres y amigos,
por su apoyo y comprensión.

A mi tutor Sergio y Stack Overflow,
por su ayuda con el trabajo.

Resumen

La turbulencia está presente en una gran cantidad de las aplicaciones cotidianas en las que intervienen fluidos y sigue siendo a día de hoy uno de los problemas físicos más importantes a resolver de forma total. La generación de conocimiento mediante experimentos es muy limitada por lo que las simulaciones por ordenador se han convertido en el camino a seguir. Las simulaciones que proporcionan un mayor nivel de detalle se denominan DNS (*Direct Numerical Simulation*) y fueron posibles gracias al aumento en la potencia computacional. Para profundizar en la comprensión de la turbulencia es necesario hacer simulaciones cada vez mayores, que requieren una gran potencia de cálculo únicamente proporcionable por superordenadores con un gran número de procesadores y haciendo uso de técnicas de computación paralela (OpenMP y MPI). Uno de los aspectos que más recursos consumen son las comunicaciones entre procesadores, cuyo coste temporal puede llegar a ser un 60 % del total, por lo que optimizarlas reduciría la cantidad de tiempo que tarda en ejecutarse la simulación, disminuyendo a su vez el coste monetario de esta.

El objetivo de este trabajo es precisamente el de optimizar el código de simulaciones DNS de canales turbulentos LISO, escrito en FORTRAN, mediante su cambio a doble precisión y la propuesta de nuevos algoritmos más rápidos para la sección de comunicaciones.

Índice general

Índice de figuras

Índice de tablas

Índice de códigos

Parte I

Memoria

Capítulo 1

Introducción

Toda persona curiosa que haya observado alguna vez el agua cayendo de una cascada o el humo saliendo de una chimenea se puede haber percatado de que el movimiento de estos fluidos no siempre es suave ni ordenado. De hecho, estos movimientos suelen ser irregulares y aparentemente aleatorios e impredecibles. Este fenómeno se conoce como turbulencia y es uno de los principios fundamentales que se esconde detrás de un gran número de aplicaciones en el ámbito de la ingeniería, sobre todo en aeronáutica. Otros ejemplo más técnicos, pero igualmente cotidianos, pueden ser el movimiento del aire alrededor de vehículos como automóviles, trenes o aeronaves, el transporte de líquidos por tuberías o la mezcla de aire y combustible en motores de combustión. Por tanto, su estudio y comprensión es un aspecto clave si se quieren seguir optimizando todas estas aplicaciones, ya sea por motivos económicos, de seguridad o de conservación del medio ambiente.

La turbulencia es un fenómeno que a lo largo de la historia ha fascinado a las mentes más brillantes de este planeta, muchas de las cuales siguen trabajando en su comprensión. Más adelante se presentará una pequeña reseña histórica, pero es sabido que las ecuaciones que rigen el comportamiento de los fluidos son ya conocidas, son las ecuaciones de Navier-Stokes, formuladas a mediados de siglo XIX y mostradas en su forma vectorial en (1.1).

$$\rho\frac{D\vec{u}}{Dt} = -\nabla p + \mu\nabla^2\vec{u} + \rho\vec{f} \tag{1.1}$$

Estas ecuaciones, pese a ser ampliamente utilizadas hoy en día ya sea con simplificaciones o mediante resolución numérica, son la base de uno de los famosos 7 *Problemas del Milenio*[1] del Instituto Clay de Matemáticas [1]. La resolución de cada problema supone una recompensa de 1 Millón de dólares ($), y el de estas ecuaciones, que presentan un reto de carácter matemático, es que aún queda por demostrar la existencia, unicidad y regularidad de sus soluciones, de ahí que se siga trabajando en ellas aunque, parece ser, con poco éxito.

Dejando a un lado el aspecto teórico y pasando al práctico, su resolución se ha conseguido de forma analítica para casos muy sencillos y de forma numérica para casos más generales, tanto de ámbito investigador como ingenieril, y ha sido posible gracias al gran desarrollo de los ordenadores en las últimas décadas, cuya potencia de cálculo no para de crecer aunque, como se verá más adelante, la tendencia está cambiando. Lo que antaño resultaba en una simulación de varios meses hoy en día se puede hacer en horas.

[1]Uno de los cuales ya ha sido resuelto, y su recompensa rechazada.

Sin embargo, pese a que conocemos las ecuaciones que dominan el comportamiento de los fluidos, no se consigue comprender de manera completa el fenómeno de la turbulencia. Es por esto que se sigue y seguirá investigando sobre ella, y es que la investigación es la única manera de obtener respuestas. Las simulaciones con este fin que se realizan en la actualidad emplean multitud de recursos computacionales y generan cantidades ingentes de datos para analizar y con los que obtener información, todo para intentar avanzar en la comprensión de este fenómeno. Un buen ejemplo de esto son los códigos de simulación de canales turbulentos, muy estudiados en los últimos años y objeto de este trabajo.

Los códigos de este tipo realizan lo que se denomina una Simulación Numérica Directa, o DNS por sus siglas en inglés, que es el método que más nivel de detalle proporciona para estudiar la turbulencia a nivel computacional y en la que se utilizan herramientas numéricas y computacionales muy específicas y cuidadas al detalle que buscan llevar a la máquina donde se ejecute al límite y exprimir toda su capacidad de cálculo. Su simulación en los supercomputadores más grandes del mundo es difícilmente accesible y muy costosa, tanto en tiempo como en dinero, por lo que cualquier optimización del código, por pequeña que parezca, puede suponer realizar una simulación mayor al mismo coste o la original de forma más barata.

El código con el que se ha tratado para la realización de este proyecto, escrito en FORTRAN y al que llaman LISO, ha sido cedido por el tutor y utilizado en algunas de las simulaciones de turbulencia de pared más grandes del mundo. De hecho, él mismo consiguió realizar la que fue la más grande del mundo durante varios años [2] y la que actualmente lo será [3].

1.1. Objetivos

El objetivo de este proyecto es el de tratar los aspectos computacionales de este tipo de simulaciones, considerados prácticamente igual de importantes que los aspectos numéricos y físicos de las mismas. Estos dos últimos aspectos se tratarán de manera más general y sintetizada, indicando las referencias en caso de que se quiera ampliar la información, buscando así tener una visión global de todo el proceso.

Dado el carácter extremo de estas simulaciones, que buscan utilizar la mayor cantidad de recursos computacionales posibles de forma eficiente, no resulta extraño que los aspectos computacionales sean muy parecidos a los de las simulaciones punteras de otras ramas de la ciencia como pueden ser la astrofísica, la biología o la física nuclear, por lo que es una manera de familiarizarse con el entorno aplicado de la computación de altas prestaciones.

Como parte más práctica, se intentarán optimizar algunos aspectos del código, buscando mejorar su precisión de resolución y reducir el tiempo de ejecución mediante nuevos algoritmos relacionados con la parte de las comunicaciones entre procesadores.

Así pues, se pretende con la realización del presente trabajo:

- Presentar una visión global de los diferentes fundamentos que están presentes en las simulaciones DNS de canales turbulentos: físicos, numéricos y computacionales.

- Destacar los aspectos computacionales más importantes del código, que afectan de manera considerable al rendimiento de la simulación así como los fundamentos de computación de altas prestaciones que los hacen posible.
- Transformar el código a precisión doble y analizar las consecuencias de dicho cambio, tanto en tiempo de ejecución como en memoria, así como sus posibles beneficios. Este cambio es necesario para poder seguir aumentando el número de Reynolds en simulaciones futuras sin perder precisión en ciertas operaciones.
- Proponer alternativas de optimización de uno de los algoritmos del código, el de las comunicaciones. Comparando posteriormente su eficacia con la versión original.

1.2. Estructura

El trabajo está dividido en tres partes. La primera corresponde a la memoria y en ella se encuentra el grueso del trabajo. Esta, a su vez, está dividida en siete capítulos:

- **Capítulo 2**: Se desarrolla de manera teórica el concepto de turbulencia, centrándose en los flujos de pared turbulentos y las diferentes técnicas de resolución.
- **Capítulo 3**: Se profundiza en las simulaciones DNS de canales turbulentos, tanto en sus fundamentos numéricos como en la técnica implementada en el código para la resolución.
- **Capítulo 4**: Se abordan los conceptos relacionados con la computación de altas prestaciones que afectan a este tipo de simulaciones costosas, presentando los recursos que se utilizan tanto a nivel *hardware* como *software*.
- **Capítulo 5**: Se exponen los aspectos computacionales que más relevancia tienen en el rendimiento del código.
- **Capítulo 6**: Se detalla la metodología utilizada para la realización del trabajo así como las optimizaciones llevadas a cabo y se analizan los resultados obtenidos comparándolos con los de las versiones originales.
- **Capítulo 7**: Se sintetiza el desarrollo del trabajo, llegando a la conclusión de si finalmente se ha conseguido o no optimizar el código.

La segunda parte corresponde al pliego de condiciones, donde se presentan brevemente las condiciones generales y particulares que aplican a este trabajo así como los recursos materiales utilizados y las etapas del propio proyecto. También se incluye el presupuesto, donde se tiene en cuenta el coste monetario de los recursos humanos y materiales.

Finalmente, en la tercera parte, se incluyen los anexos que procedan.

Capítulo 2

Turbulencia

La turbulencia no es una propiedad de un fluido sino un estado del flujo y, aunque de difícil definición exacta, es un fenómeno determinista, no aleatorio, aunque altamente caótico, no homogéneo, tridimensional, no lineal y no estacionario, difusivo en sus escalas grandes (llegando a tamaños planetarios como en el gaseoso Júpiter, ver Figura 2.1 [4]) y disipativo en las pequeñas, que lleva siendo objeto de estudio durante muchos siglos. Si se vuelve la vista atrás, se puede decir que posiblemente fue Leonardo da Vinci, alrededor del siglo XVI, una de las primeras personas que se interesó por el estudio de la turbulencia, y buena fe de ello dan sus ilustraciones relacionadas con la caída del agua. A él le siguieron nombres como Hagen o Darcy, que a mediados del siglo XIX empezaron a distinguir dos tipos de flujo gracias a su estudio del transporte de agua en largas tuberías. Más tarde, Boussinesq [5] publicó un artículo donde se distinguían ambos tipos de flujo y se presentaban hipótesis de posibles explicaciones de sus comportamientos. Pero no fue hasta finales de siglo cuando, gracias a los experimentos de Reynolds [6] y a la definición de una magnitud que hoy lleva su nombre, se concretó una manera clara, matemática, de definir cuando un flujo era suave (laminar) o irregular (turbulento).

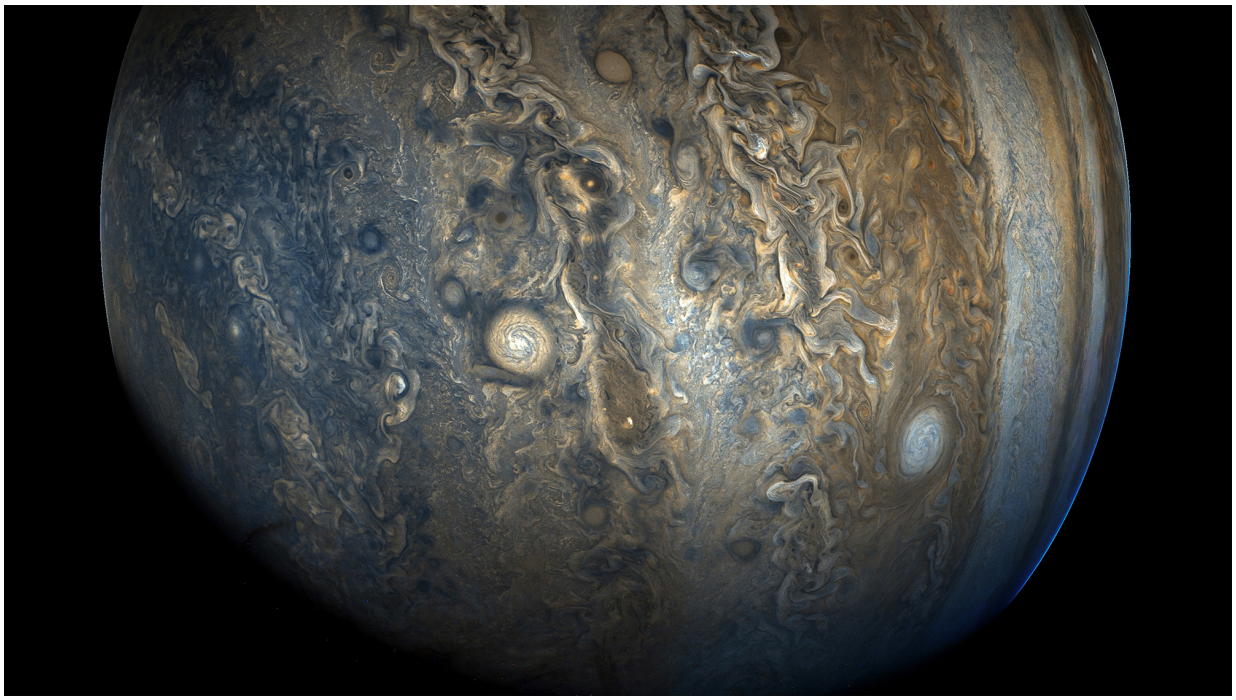


Figura 2.1: Estructuras turbulentas en Júpiter captadas por la sonda Juno de la NASA.

2.1. Tipos de flujo

Esta magnitud adimensional, el Número de Reynolds (2.1), representa la relación entre las fuerzas inerciales y las viscosas en el fluido y es la que permite determinar la naturaleza de un flujo. Su valor puede diferir en varios órdenes de magnitud dependiendo de la aplicación considerada.

$$Re = \frac{\rho u L}{\mu} = \frac{uL}{\nu} \tag{2.1}$$

donde ρ es la densidad del fluido, u es la velocidad del fluido con respecto al cuerpo, L es la longitud característica del problema, μ es la viscosidad dinámica del fluido y $\nu = \mu/\rho$ es la viscosidad cinemática del fluido.

Un Reynolds bajo indica que las fuerzas viscosas dominan sobre las inerciales, el flujo es suave, laminar. De forma análoga, si las fuerzas inerciales dominan sobre las viscosas el flujo es irregular, turbulento. Esta diferencia de regímenes, según el propio Reynolds, ocurre a $Re \approx 2 \cdot 10^3$ para flujos internos en tuberías y a $Re \approx 2 \cdot 10^5$ para flujos externos sobre una placa plana. Sobre todo en este último caso, de suma importancia en aplicaciones aeronáuticas, se puede también definir un Número de Reynolds local Re_x, donde la longitud característica x va aumentando desde el comienzo del dominio en la dirección del movimiento del fluido. Así, en momentos iniciales el flujo suele ser laminar pues x tiene un valor pequeño y a medida que el flujo avanza transiciona a turbulento. Una esquemática de este proceso de transición puede observarse en la Figura 2.2 [7].

Esta transición es también objeto de estudio exhaustivo, ya que un gran porcentaje de la resistencia al avance de vehículos como aeronaves o automóviles, que afecta de manera considerable a su consumo de combustible, es debida a la alta fricción entre el fluido y sus superficies externas causada por una capa límite (región del espacio desde la superficie del cuerpo hasta el flujo libre) turbulenta. Por ello, cualquier avance que permita reducir esta fricción mejorará de forma considerable la eficiencia de estos vehículos.

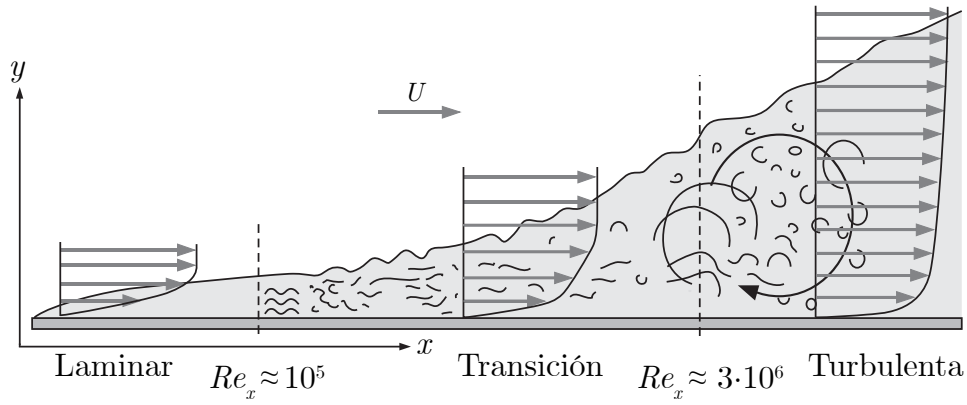


Figura 2.2: Transición de una capa límite laminar a turbulenta.

2.2. Flujos de pared

En ingeniería muy pocos flujos son laminares, principalmente aquellos relacionados con la lubricación debido a la alta densidad de estos fluidos, la mayoría son turbulentos. Como flujos externos encontramos aquellos en los que un fluido envuelve a un cuerpo, principalmente vehículos, tanto aéreos, terrestres como acuáticos, pero también en otras aplicaciones como molinos de viento. Sin embargo, el flujo turbulento interno también está muy presente, principalmente relacionado con plantas propulsoras de combustión y sus gases de escape. La relación de todas estas aplicaciones es que hay paredes presentes, bien una pared en el caso externo o dos en el interno, lo que condiciona el comportamiento del fluido. Cabe destacar que otras aplicaciones turbulentas, tanto internas como externas, pueden no cumplir esta condición, un ejemplo es el caso de los chorros.

Con la presencia de paredes y, debido a que los fluidos comunes presentan una viscosidad no nula, se cumple la condición de que sobre la superficie de la pared el vector velocidad del fluido y de la pared deben ser iguales (condición de no deslizamiento). Se pueden distinguir dos tipos de flujo de este tipo, el de Couette y el de Poiseuille (o una combinación de ambos), que son precisamente los dos casos que el código de simulación LISO es capaz de resolver. Estos dos casos presentan soluciones analíticas para regímenes laminares e incompresibles mediante simplificaciones de las ecuaciones de Navier-Stokes (1.1). En el caso ideal ambos flujos transcurren entre dos paredes infinitas paralelas separadas por una distancia que se mantiene fija, lo que se conoce como canal. En el flujo de Couette una de las dos paredes está en movimiento a velocidad U_w y el fluido se mueve debido a la condición de no deslizamiento en ella. Para el flujo de Poiseuille las paredes no se mueven sino que es un gradiente de presión el que ocasiona el movimiento del fluido, que se mueve hacia la región de menos presión. Se puede observar una representación de ambos flujos en la Figura 2.3 [7].

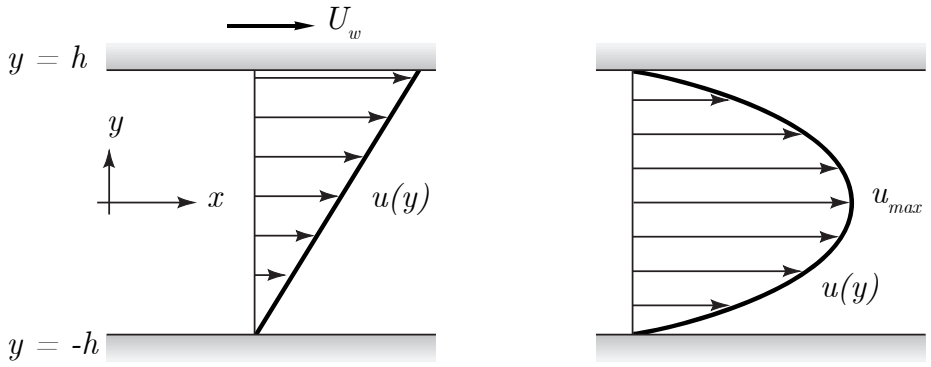


Figura 2.3: Flujos de Couette y Poiseuille.

Debido a la condición de no deslizamiento aparecen esfuerzos viscosos tangenciales τ_w en la pared y, a partir de ellos, se puede definir una velocidad de fricción u_τ tal y como se puede observar en (2.2).

$$u_\tau^2 = \frac{\tau_w}{\rho} = \nu \left.\frac{\partial u}{\partial y}\right|_w \tag{2.2}$$

donde u es el módulo de la velocidad en dirección paralela a la pared, la componente y indica dirección normal a la pared y el subíndice w representa sobre la superficie de la pared.

A partir de la velocidad de fricción se puede definir un Número de Reynolds de fricción (2.3) que, como se verá más adelante, es uno de los indicadores que permite caracterizar el tamaño de las simulaciones en términos de coste computacional así como la naturaleza y características del flujo a estudiar.

$$Re_\tau = \frac{u_\tau h}{\nu} \tag{2.3}$$

donde la longitud característica h es la mitad de la altura del canal.

2.3. Capa límite

A partir de la velocidad de fricción (2.2) se pueden definir una velocidad y distancia a la pared de forma adimensional (2.4) que facilitan el estudio del flujo cerca de la pared, donde existe una región llamada capa límite (Figura 2.2) con un gradiente de velocidades entre la superficie del cuerpo y el flujo libre debido a la viscosidad no nula del fluido de manera que la velocidad media del flujo depende de la distancia a la pared.

$$u^+ = \frac{u}{u_\tau} \qquad y^+ = \frac{y u_\tau}{\nu} \tag{2.4}$$

La distancia de pared y^+ se puede entender como un símil al Número de Reynolds local, ya que determina la importancia entre los procesos turbulentos y viscosos. La velocidad adimensional u^+ depende de y^+ de diferente forma según la región de la capa límite que se examine, las llamadas leyes de pared. A valores bajos, $y^+ \leq 5$, cerca de la pared, la pendiente $\partial u^+/\partial y^+$ es lineal ($u^+ \approx y^+$). A esta región se le conoce como subcapa viscosa y en ella los esfuerzos cortantes turbulentos son despreciables frente a los viscosos. A valores más altos, $y^+ \geq 70$ y hasta $y/h \leq 0{,}2$, el comportamiento se rige por la ley logarítmica de Von Kármán (2.5), por lo que a esta región se le llama subcapa logarítmica y en la que los esfuerzos cortantes turbulentos dominan sobre los viscosos.

$$u^+ = \frac{1}{\kappa} \log y^+ + C \tag{2.5}$$

donde κ es la constante de Von Kármán, y C otra constante. El valor de ambas constantes depende del problema y las condiciones y se extraen de experimentos físicos y de simulaciones DNS como la presente.

Por último, entre estas dos regiones existe una a la que se le denomina subcapa *buffer*, que no se rige por ninguna ley ya que hace de nexo entre las dos leyes anteriores. En la Figura 2.4 se puede observar la representación de las leyes de pared de las subcapas comentadas. Más información sobre la capa límite y sus subcapas puede encontrarse en Pope [8].

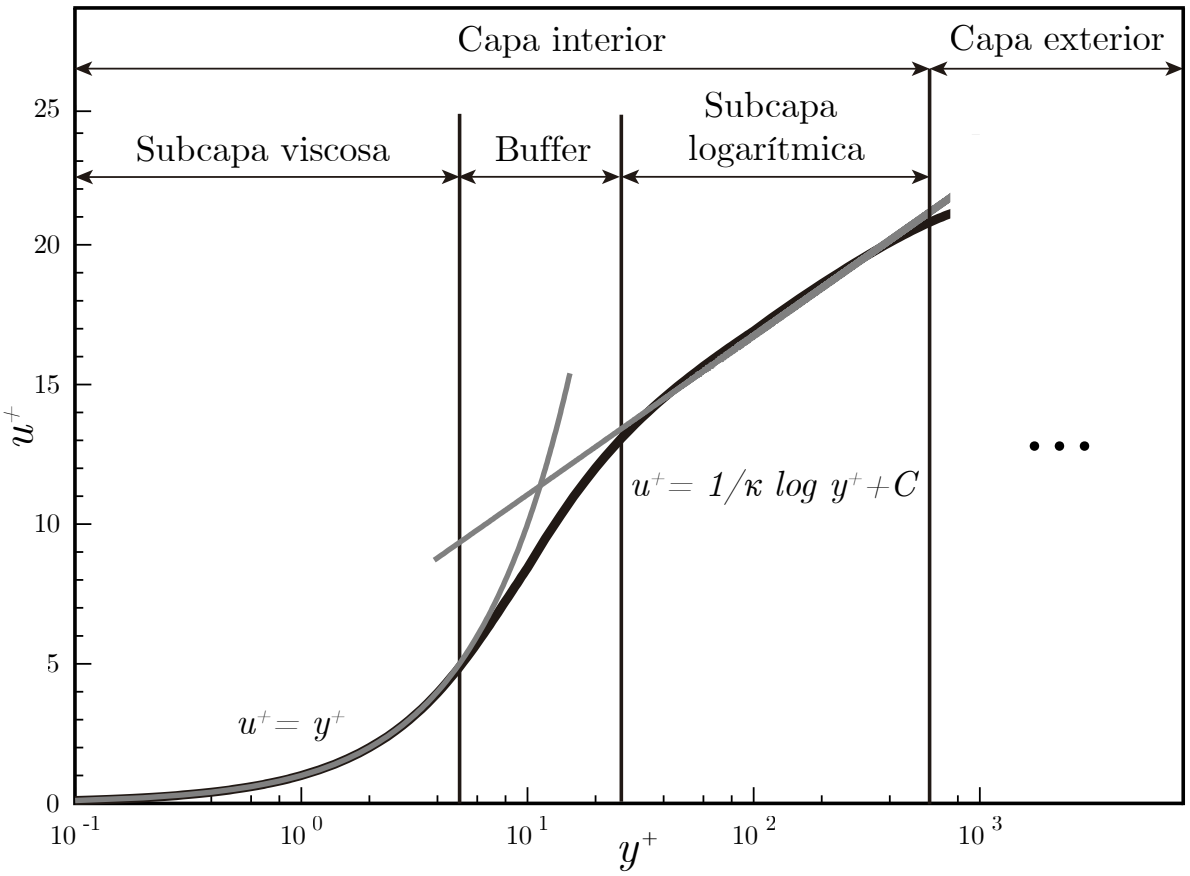


Figura 2.4: Leyes de pared para una capa límite.

2.4. Escalas turbulentas y cascada de energía

Siguiendo con el hilo histórico, fue Richardson, un meteorólogo, cuando a principios del siglo XX empezó a aplicar los estudios de turbulencia a su rama para intentar predecir el tiempo y se dio cuenta, tal y como confirmó Kolmogorov [9] más tarde, de que existían estructuras turbulentas de tamaños muy distintos, desde los más grandes a los más pequeños. Los estudios de Kolmogorov son considerados como algunos de los más importantes en el avance de la comprensión de la turbulencia. Según su teoría, existen estructuras turbulentas o torbellinos de varios tamaños diferentes o escalas.

Por un lado se encuentra la escala de longitud más grande o integral $\mathcal{L}$, del orden de la longitud característica del problema, donde el Reynolds es alto, la viscosidad despreciable y en la que se inyecta energía al fluido. Por otro lado encontramos las escalas más pequeñas o de Kolmogorov η (2.6), isótropas e independientes del problema y donde domina la viscosidad, que determina el tamaño de estas escalas y que consigue disipar estos torbellinos. Entre medias hay un rango de escalas ℓ en las que los torbellinos de escala integral se van rompiendo progresivamente en otros de menor escala, transfiriéndoles así parte de su energía, hasta llegar a la escala de Kolmogorov, donde se disipan en calor debido a los esfuerzos cortantes viscosos. Este fenómeno se conoce como cascada de energía.

$$\eta = \left(\frac{\nu^3}{\varepsilon}\right)^{1/4} \tag{2.6}$$

donde ε es la tasa de disipación de energía.

La energía turbulenta (E) representa la energía cinética media presente en los torbellinos de un flujo turbulento y se distribuye a lo largo del rango de escalas. Se puede así definir un espectro de energía (su distribución) dependiendo de la escala de los torbellinos. En las escalas integrales se introduce energía al sistema, por lo que esta va aumentando hasta llegar a una cantidad máxima de energía. Estas escalas corresponden a frecuencias ($\omega = 2\pi/T$, siendo T el periodo del movimiento) bajas por lo que su número de onda ($k = 2\pi/l$, siendo l el tamaño de la escala espacial), que es la propiedad equivalente a la frecuencia empleando el dominio espacial en lugar del temporal, es también pequeño. A medida que estos torbellinos se van rompiendo su tamaño disminuye y su número de onda aumenta hasta llegar a la escala de Kolmogorov, donde su número de onda es máximo y su energía mínima pues acabará disipándose. Este fenómeno se puede observar en la Figura 2.5 [10].

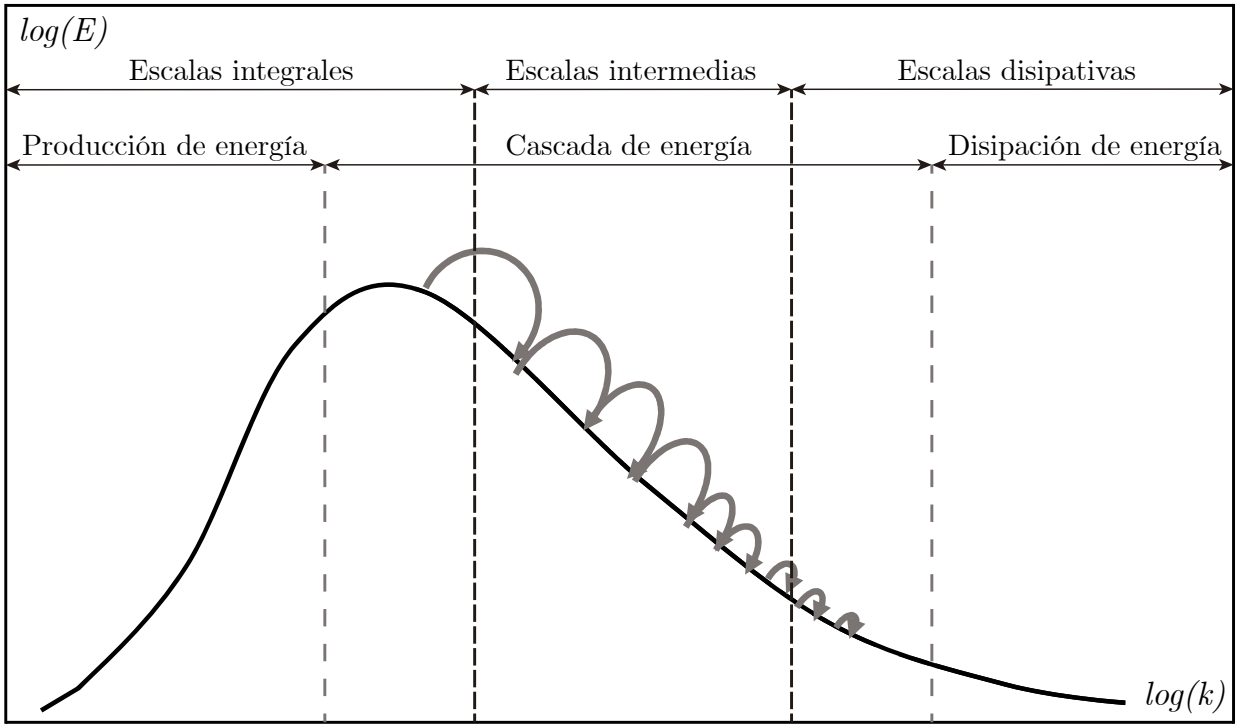


Figura 2.5: Representación de la cascada de energía.

Como se señalaba en el Capítulo 1, no es el objetivo de este trabajo el de explicar en detalle los fundamentos físicos de la turbulencia o teorías más completas por lo que se refiere al lector a otras fuentes como Pope [8] o Davidson [10] para más información.

2.5. Mecánica de fluidos computacional

Como se ha expuesto a lo largo del capítulo, los flujos turbulentos no pueden ser resueltos con precisión de manera analítica, por lo que se ha de recurrir a métodos numéricos que utilicen herramientas computacionales para ello. Esto ha sido posible debido a los avances en la potencia computacional de los ordenadores. Hasta ahora, esta potencia se ha ido multiplicando por 1000 cada 15 años [11], por lo que simulaciones imposibles de realizar en el pasado ahora pueden ser triviales. En el Capítulo 4 se profundizará en estos aspectos.

Además, se acaba de afirmar en la sección anterior que la turbulencia está presente en las múltiples escalas de un problema, por lo que se tendrá que marcar el nivel de detalle que se desea. Existen simulaciones numéricas de distinto nivel de detalle, dependiendo de la aplicación deseada, pero la norma es que cuanto más detalle y precisión se desee más computacionalmente costosas serán. Se pueden distinguir, con este criterio, tres tipos o esquemas de simulaciones bien diferenciadas:

2.5.1. *Reynolds-Averaged Navier-Stokes* (RANS)

Este esquema utiliza la ayuda de las técnicas estadísticas de manera que las magnitudes del fluido, como la velocidad o la presión, se pueden expresar mediante un valor medio y un valor de desviación o fluctuación (2.7), lo que se conoce como descomposición de Reynolds, siendo la media de las fluctuaciones nula por definición.

$$\phi = \widehat{\phi} + \phi' \tag{2.7}$$

donde ϕ es el valor de la magnitud en un instante dado, $\widehat{\phi}$ su valor medio y ϕ' su fluctuación en ese instante.

Esta descomposición de Reynolds de las magnitudes del fluido se aplica a las ecuaciones originales de Navier-Stokes (1.1) y a la de continuidad con tal de poder resolverlas y, para flujo incompresible, expresadas mediante el convenio de suma de Einstein y despreciando las fuerzas volumétricas, se llega a las ecuaciones de Navier-Stokes promediadas (2.8), de ahí su nombre: *Reynolds-Averaged Navier-Stokes* (RANS).

$$\frac{\partial \widehat{u_i}}{\partial t} + \widehat{u_j}\frac{\partial \widehat{u_i}}{\partial x_j} = -\frac{1}{\rho}\frac{\partial \widehat{p}}{\partial x_i} + \nu\frac{\partial}{\partial x_j}\left(\frac{\partial \widehat{u_i}}{\partial x_j} + \frac{\partial \widehat{u_j}}{\partial x_i}\right) - \frac{\partial}{\partial x_j}(\widehat{u_i'u_j'}) \tag{2.8}$$

donde el último término $\widehat{u_i'u_j'} = R_{ij}$ se conoce como el tensor de esfuerzos de Reynolds.

Finalmente se llega a un estado en el que se tienen 4 ecuaciones (1 de continuidad y 3 de momento) y 10 incógnitas (3 velocidades, la presión y los 6 componentes de R_{ij} que es simétrico), por lo que no se puede resolver. Este problema se conoce como el problema de cierre y para poder resolver las ecuaciones se hace necesario modelar el tensor de Reynolds. Una de las opciones es modelar la totalidad del tensor de Reynolds, la otra, más común, fue propuesta por Boussinesq [5], cuya hipótesis establecía que el transporte de momento se podía modelar mediante una magnitud llamada viscosidad cinemática turbulenta ν_t. Con esta hipótesis y descomponiendo el tensor de Reynolds en sus partes isótropa y anisótropa las ecuaciones RANS (2.8) son similares en forma a las estándares Navier-Stokes (1.1).

Para completar el problema de cierre solo queda determinar el valor de la viscosidad turbulenta, que no se conoce. Esto es precisamente en lo que se basan la mayoría de códigos RANS, que modelan la viscosidad turbulenta mediante ecuaciones de transporte extra como los modelos de Spalart-Allmaras, $k-\varepsilon$, $k-w$... cada uno óptimo para un tipo de aplicación y con descripciones numéricas distintas. Este método se suele utilizar en régimen estacionario, aunque ciertas modificaciones permiten resolver también el no estacionario (*Unsteady*-RANS).

El RANS es el esquema que proporciona un menor nivel de detalle ya que no se resuelve ninguna escala de la turbulencia sino que se modelan. Por ser el esquema más simple este presenta un bajo coste computacional, pero que sea menos detallado no significa que no sea útil. De hecho, las simulaciones RANS son las más utilizadas en aplicaciones de ingeniería en el ámbito industrial. Esto es porque el bajo coste computacional y su precisión suficiente para esas aplicaciones permite realizar un número grande de simulaciones en bajo tiempo en las que hacer estudios paramétricos para evaluar la efectividad de cierta característica de interés, cuya forma o naturaleza se va variando en cada simulación.

2.5.2. *Large-Eddy Simulation* (LES)

El esquema LES es el intermedio a nivel de detalle, ya que se resuelven las escalas de turbulencia más grandes pero no las pequeñas, que se aíslan de las anteriores mediante un filtrado espacial y que se modelan con métodos similares a los del esquema RANS. Al resolver las escalas grandes la simulación se debe hacer en régimen no estacionario, siendo así más precisa en los casos en los que estas escalas grandes son las dominantes. Como bien se señalaba anteriormente, una mayor precisión conlleva un mayor gasto computacional, por lo que los modelos LES no se utilizan tanto en la industria pero sí en otras aplicaciones donde las escalas grandes tienen un papel importante, como la meteorología.

2.5.3. *Direct Numerical Simulation* (DNS)

Por último se encuentran las simulaciones DNS, explicadas con más profundidad en el siguiente capítulo, que son las que cuentan con un mayor nivel de detalle ya que en ellas se resuelven todas las escalas de la turbulencia, desde las integrales hasta las de Kolmogorov, sin emplear modelos adicionales. Este tipo de simulación aunque simple, pues únicamente se trata de resolver las ecuaciones de Navier-Stokes de forma numérica sin simplificaciones, es muy costosa computacionalmente ya que las celdas del dominio a simular deben poder contener las escalas más pequeñas en su interior, por lo que es necesario una malla muy grande para un espacio físico real pequeño. Se suelen realizar para geometrías sencillas por lo que tienen un interés investigador, buscando tener una mayor comprensión del fenómeno de la turbulencia. Entre estas geometrías destacan los canales (Figura 3.1) y las tuberías para el estudio de turbulencia desarrollada y el de placas planas para el estudio de la transición de una capa límite laminar a una turbulenta, como se veía anteriormente (Figura 2.2).

Una comparación de estos esquemas comentados se puede observar en la Figura 2.6 [12], donde se observa una esquemática idealizada del nivel de detalle de la resolución de los diferentes esquemas numéricos para un esfera ante una corriente de aire, siendo posible visualizar las estructuras turbulentas en las más detalladas, y en la Figura 2.7, donde se muestran las tendencias de algunas de las características más importantes presentes en los diferentes tipos de simulación.

Figura 2.6: Detalle de resolución de las diferentes técnicas de simulación CFD.

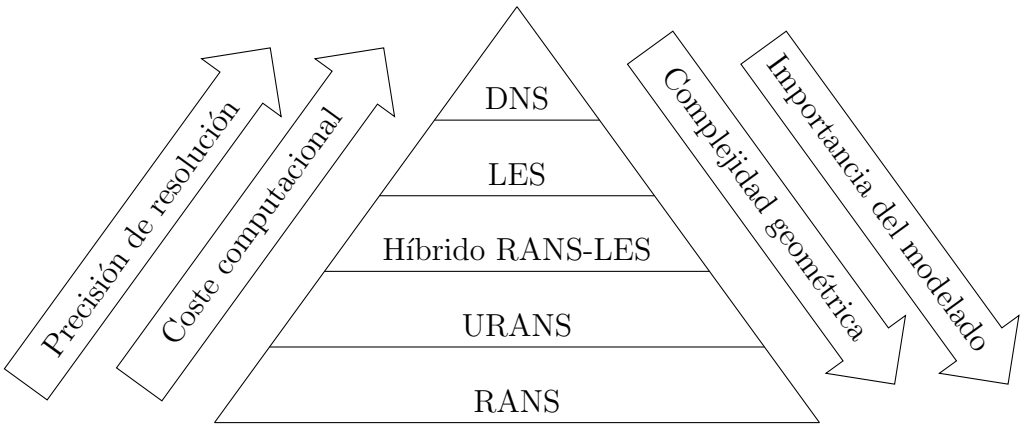


Figura 2.7: Características de las diferentes técnicas de simulación CFD.

Capítulo 3

Simulación DNS de canales turbulentos

Las simulaciones DNS fueron inviables hasta la década de los 70, cuando los ordenadores alcanzaron un poder computacional considerable, hecho que aprovecharon otras ramas de la ciencia, no solo la mecánica de fluidos, para dar pasos agigantados en la creación de nuevo conocimiento. Como se ha comentado en el capítulo anterior, este tipo de simulaciones son muy caras computacionalmente ya que, incluso para un dominio físicamente pequeño, el número de puntos de mallado en los que discretizar y resolver las ecuaciones de Navier-Stokes es muy grande debido a que es necesario capturar de forma precisa las escalas más grandes y más pequeñas de las estructuras turbulentas.

La primera simulación de este tipo fue realizada en 1972 por Orszag y Patterson [13] para turbulencia isótropa en una caja con con periodicidad en las tres dimensiones. La caja tenía 32 puntos de discretización por lado, que hacían un total de $3 \cdot 10^4$ puntos, y un Número de Reynolds de $Re_\lambda = 35$, donde λ representa la microescala de longitud de Taylor, de tamaño intermedio entre las escalas integrales y las disipativas. Se suelen utilizar como indicativo Re_λ para los casos de turbulencia isótropa y Re_τ para los de anisótropa, que son la mayoría. Para más información sobre ambos casos referirse a Pope [8] o Jiménez [14].

El caso de los canales turbulentos, que son los que el código LISO puede resolver, son más complicados ya que la turbulencia es anisótropa debido a la presencia de las paredes. La primera DNS de este tipo fue realizada en 1987 por Kim, Moin y Moser [15] para un valor de $Re_\tau = 180$ usando $4 \cdot 10^6$ puntos de malla física. Como veremos en la última sección, este valor de Re_τ esta dos órdenes de magnitud por debajo de las simulaciones punteras a día de hoy. Los autores de esta pionera simulación eran todos ellos ingenieros que pertenecían al Centro de Investigación NASA Ames, que tenía en su día como una de sus prioridades el estudio de la dinámica de fluidos y la turbulencia debido a sus importantes aplicaciones en el ámbito aeroespacial.

El coste computacional se puede medir, tanto en velocidad como en memoria, mediante el número de puntos de mallado, el de pasos temporales y el de operaciones a realizar necesarias para poder resolver el flujo, todos proporcionales y con dependencia no lineal al Número de Reynolds, ya sea Re_λ para la isótropa como Re_τ para la anisótropa. Un resumen de estas proporcionalidades (que no igualdades) dependiendo del tipo de tur-

bulencia puede encontrarse en la Tabla 3.1 [8][11]. Para entrar en contexto, el valor del Número de Reynolds de fricción del flujo sobre un pájaro volando es de $Re_\tau \approx 200$ y sobre un avión de tamaño medio de $Re_\tau \approx 40000$ [11].

Turbulencia	Puntos	Pasos temporales	Operaciones
Isótropa	$0{,}06Re_\lambda^{4,5}$	$9{,}2Re_\lambda^{1,5}$	$0{,}55Re_\lambda^{6}$
Anisótropa	$Re_\tau^{2,7}$	$40Re_\tau^{1,05}$	$Re_\tau^{3,75}$

Tabla 3.1: Proporcionalidad del coste computacional según el tipo de turbulencia.

Este tipo de simulaciones, pese a su coste temporal y monetario, son perfectas para el ámbito investigador ya que permiten conocer toda la información del campo fluido en cualquier instante, lo que con técnicas experimentales no es posible, bien porque los instrumentos no puedan medir todas las variables deseadas o bien porque estos instrumentos perturben de alguna manera el flujo a estudiar. Además, todos los datos obtenidos son almacenados y pueden ser compartidos y analizados tantas veces como se desee. En el siguiente capítulo se hablará del tamaño de los datos generados y del problema que supone su procesado y almacenamiento.

3.1. Dominio

El dominio del problema, que se puede observar en la Figura 3.1, es el mismo independientemente de si se quiere simular un caso con flujo de Couette o de Poiseuille, ya que únicamente cambian las condiciones de contorno. La altura del canal es $2h$ y las longitudes en dirección de la corriente y de profundidad son L_x y L_z respectivamente. Estas dos últimas dimensiones son las consideradas periódicas en múltiplos de 2π por lo que se le asignan los valores de $L_x = \frac{2\pi}{\alpha}h$ y $L_z = \frac{2\pi}{\beta}h$, donde se varían los parámetros α y β, proporcionales a los números de onda k_x o k_z en sus respectivas direcciones.

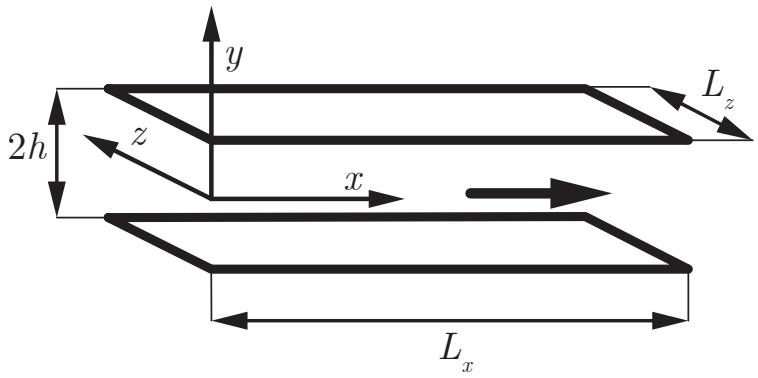


Figura 3.1: Geometría del canal turbulento, dimensiones y sistema de coordenadas.

3.2. Resolución

El código simula casos con flujo incompresible de canales turbulentos utilizando el esquema numérico desarrollado por Kim, Moin y Moser [15], al que se recomienda acudir en caso de querer profundizar en el desarrollo matemático, pues se comentará de manera simplificada. Este procedimiento se fundamenta en la transformación de las 4 ecuaciones de conservación[1] (una de conservación de la masa (3.1) y tres de la conservación de la cantidad de movimiento (3.2), las de Navier-Stokes) en dos ecuaciones de transporte de mayor orden basadas en dos incógnitas, la vorticidad en dirección normal a la pared ω_y y el Laplaciano de la velocidad normal a la pared $\phi = \nabla^2 v$, que son las que se resuelven. Más adelante se comentará que esta selección de variables no es trivial y que conlleva beneficios en el coste computacional.

$$\frac{\partial u_i}{\partial x_i} = 0 \tag{3.1}$$

$$\frac{\partial u_i}{\partial t} = -\frac{\partial p}{\partial x_i} + H_i + \frac{1}{Re_\tau}\nabla^2 u_i \tag{3.2}$$

donde se puede definir una nueva magnitud $\vec{H}$ (3.3) considerada como una helicidad modificada y que incluye los términos convectivos y de presión.

$$\vec{H} = (\vec{u} \times \vec{\omega}) - \frac{1}{2}\nabla(\vec{u} \cdot \vec{u}) \tag{3.3}$$

donde las componentes de cada vector se definen como: $\vec{u} = (u, v, w)$, $\vec{\omega} = (\omega_x, \omega_y, \omega_z)$ y $\vec{H} = (H_1, H_2, H_3)$.

Finalmente (se omiten los pasos intermedios, referirse a [15] para el desarrollo completo) se llega a las dos ecuaciones de transporte que antes se comentaban (3.4) y (3.5).

$$\frac{\partial \phi}{\partial t} = h_v + \frac{1}{Re_\tau}\nabla^2 \phi \tag{3.4}$$

$$\frac{\partial \omega_y}{\partial t} = h_g + \frac{1}{Re_\tau}\nabla^2 \omega_y \tag{3.5}$$

donde h_v y h_g se definen conforme a (3.6) y (3.7).

$$h_v = \left(\frac{\partial^2}{\partial x^2} + \frac{\partial^2}{\partial z^2}\right) H_2 - \frac{\partial}{\partial y}\left(\frac{\partial H_1}{\partial x} + \frac{\partial H_3}{\partial z}\right) \tag{3.6}$$

$$h_g = \frac{\partial H_1}{\partial z} - \frac{\partial H_3}{\partial x} \tag{3.7}$$

y que, junto a (3.8) (3.9) (3.10) (3.11) y (3.12), se completa el sistema de ecuaciones a resolver.

$$\nabla^2 v = \phi \tag{3.8}$$

[1] Sin contar la de la energía, aunque otras versiones de LISO ya la implementan para resolver casos con transferencia de calor [16].

$$\frac{\partial u}{\partial x} + \frac{\partial w}{\partial z} = -\frac{\partial v}{\partial y} \tag{3.9}$$

$$\frac{\partial u}{\partial z} - \frac{\partial w}{\partial x} = \omega_y \tag{3.10}$$

$$\frac{\partial \omega_x}{\partial x} + \frac{\partial \omega_z}{\partial z} = -\frac{\partial \omega_y}{\partial y} \tag{3.11}$$

$$\frac{\partial \omega_z}{\partial x} - \frac{\partial \omega_x}{\partial z} = \phi \tag{3.12}$$

Si uno se da cuenta, el término de presión p que aparecía en las ecuaciones iniciales (3.2) no aparece en el sistema de ecuaciones finales de forma explícita. Sin embargo, puede ser obtenido por pos-procesado a partir de las demás variables. Precisamente el hecho de que no haya que resolver el campo de presiones es una ventaja, que, unida al hecho de que solo se calculan dos ecuaciones de transporte y no una para cada velocidad, hace que el código sea más eficiente. Únicamente la elección de esas dos incógnitas, ϕ y ω_y, permite estas simplificaciones. Para más información sobre la resolución del sistema de ecuaciones planteado y la recuperación de variables físicas referirse, una vez más, a Kim, Moin y Moser [15].

Al igual que en otros problemas de la física, para resolver este problema se utiliza un método pseudo-espectral, basado en transformadas discretas de Fourier por su bajo coste computacional, que considera las direcciones paralelas a la pared periódicas en sus fronteras debido a que en estas dos direcciones se puede considerar flujo homogéneo y desarrollado además del hecho de que el concepto de paredes infinitas no es implementable de forma discreta tradicional, pero no así en la dirección normal a las paredes debido a la anisotropía generada por la presencia de las mismas. En el caso de turbulencia isótropa el dominio sería periódico en las tres direcciones y se podrían utilizar métodos totalmente espectrales. El adjetivo *pseudo* se debe a que este método presenta un dominio computacional en dos espacios diferentes, uno físico y uno de Fourier (el espectral), la razón es que los productos dobles no lineales de la helicidad (3.3) se calculan en espacio físico de manera más eficiente.

Este cambio de espacios conlleva dos consecuencias importantes desde el punto de vista computacional. En primer lugar, el espacio de Fourier se expande en un factor de 3/2 al pasarlo al físico en las dos direcciones periódicas para evitar errores de *aliasing*, producidos por las transformadas discretas de Fourier que introducen energía de los modos más altos en los más pequeños, lo que hace que sea caro en memoria. Más información sobre el problema del *aliasing* y métodos espectrales aplicados a la mecánica de fluidos en Canuto [17]. En segundo lugar, y ampliado en el Capítulo 5, tenemos el hecho de que para ser capaz de realizar la transformada discreta de Fourier en las dos direcciones periódicas de forma óptima es imprescindible tener los datos necesarios para ella guardados en la memoria local, idealmente de forma contigua. Esto hace que los distintos procesadores en los que se divide el dominio tengan que intercambiar datos, lo que se conoce como comunicaciones, que consumen una gran parte del tiempo por paso. La estructura simplificada del algoritmo con significado físico puede observarse a continuación:

Estructura física del algoritmo

1. Calcular velocidad y vorticidad en el dominio de Fourier.
2. Transformar x y z al dominio físico.
3. Calcular la helicidad.
4. Transformar x y z al dominio de Fourier.
5. Calcular la parte derecha de las dos ecuaciones de transporte.
6. Resolver los sistemas de ecuaciones.
7. Avanzar en tiempo.

3.3. Esquema numérico

Para ser capaz de resolver el sistema de ecuaciones que gobierna el problema es necesario transformar las ecuaciones diferenciales en algebraicas mediante la discretización de las derivadas tanto espaciales como temporales. De la misma manera, se necesitarán también condiciones de contorno e iniciales para poder resolverlas, lo que se detalla brevemente a continuación.

3.3.1. Condiciones de contorno

Se pueden considerar dos regiones diferentes en las que aplicar las condiciones de contorno. Por un lado las propias paredes, correspondientes a la dirección y normal a las mismas y, por otro lado, las direcciones paralelas a las paredes x, z.

Las condiciones de contorno sobre la superficie de las paredes (no permeables) varía, como se ha comentado anteriormente, de si se trata de un caso de flujo de Couette o de Poiseuille. En el primer caso la pared superior se movería con una velocidad relativa a la pared de abajo de $\vec{U_w} = (U_w, 0, 0)$, por lo que esta sería la condición de contorno del fluido en la pared superior, mientras que en la inferior, al no moverse, la velocidad del fluido sería nula debido a la condición de no deslizamiento. Para el caso de Poiseuille esta velocidad relativa U_w sería nula y se impondría el gradiente de presión en la dirección del flujo $\partial p/\partial x$. En cuanto a las direcciones paralelas a las paredes, estas se consideran periódicas en sus fronteras.

3.3.2. Condiciones iniciales

En cuanto a las condiciones iniciales, todos los casos leen el campo fluido de un archivo con datos de simulaciones anteriores de orden similar y el flujo evoluciona desde ese instante hasta pasar un estado de transición y ajustarse de manera desarrollada a las nuevas condiciones impuestas. En este archivo inicial se encuentran, entre otros, los valores en todo el canal de ϕ y ω_y y los vectores de velocidad iniciales para u y w.

3.3.3. Discretización espacial

De forma análoga a las condiciones de contorno, se puede diferenciar la discretización espacial en las direcciones normal y paralelas a las paredes.

Para las derivadas en dirección y se utiliza el método de las Diferencias Finitas Compactas de siete puntos con resolución espectral, una versión mejorada del método de Diferencias Finitas. Más información sobre este método en [18] y [19]. En el caso de las direcciones x y z, con periodicidad, se utilizarán expansiones de las variables en series truncadas de Fourier en esas direcciones.

3.3.4. Discretización temporal

Para la discretización temporal se utiliza un método semi-implícito de tres pasos de Runge-Kutta [20], muy utilizados en problemas de valor inicial.

3.3.5. CFL

La condición de Courant-Friedrichs-Lewis o CFL (3.13) establece la relación entre la discretización temporal y la espacial para asegurar la estabilidad y convergencia del esquema numérico utilizado.

$$CFL = u\frac{\Delta t}{\Delta x} \tag{3.13}$$

donde u es la velocidad, Δt el paso temporal y Δx el paso espacial.

Esta condición se utiliza para asegurar que una partícula fluida no avance más que la resolución de la malla en un paso temporal, por lo que la condición a cumplir es que $CFL \leq 1$. En en caso de LISO se utiliza $CFL = 0{,}9$. Dado que la malla del dominio computacional la introduce el usuario según sus intereses, esta condición permite obtener el paso temporal máximo a utilizar en el problema.

3.4. Estado del arte

Mucho han avanzado las simulaciones de canales turbulentos en los casos anisótropos, desde la pionera de Kim, Moin y Moser en 1987 de $Re_\tau = 180$ [15], siguiéndole otras destacadas como la de Hoyas y Jiménez en 2003 de $Re_\tau = 2000$ [2] o la de Lee y Moser en 2015 de $Re_\tau = 5200$ [21], hasta llegar a la que, a fecha de redacción de este proyecto, será (pues está en proceso) la más grande hasta la fecha, realizada por Hoyas *et al.* en 2019 de $Re_\tau = 10000$ [3]. Este valor de Re_τ se cree que será lo suficientemente alto como para poder evaluar ciertas teorías relevantes al estudio de la turbulencia en la geometría de un canal turbulento [11], por lo que posiblemente, y debido a las limitaciones computacionales actuales de incrementar el tamaño del dominio, la tendencia a seguir sea la de estudiar geometrías y flujos más complejos.

Capítulo 4

Computación de altas prestaciones

La evolución de los ordenadores desde su surgimiento a mediados del siglo XX ha sido muy rápida, tanto en su arquitectura como en su capacidad computacional. Los primeros ordenadores, alrededor de 1940, funcionaban mediante tubos de vacío y ocupaban una gran cantidad de espacio, siendo ejemplos históricamente reconocidos el Colossus, utilizado para descifrar las comunicaciones alemanas en la 2ª GM, o el ENIAC, el primer ordenador de propósito general que podía ser reprogramado para ejecutar otra tarea. Más adelante, alrededor de 1960, se sustituyeron los tubos de vacío por transistores, más pequeños y eficientes que los anteriores, que fueron una de las causas del rápido crecimiento de la potencia computacional. Los primeros transistores tenían un tamaño de unos 30 μm mientras que los más avanzados hoy en día rondan los 7 nm. Más tarde, sobre 1970, el rápido aumento de la cantidad de transistores presentes propició la aparición de los circuitos integrados, que facilitaban la fabricación e instalación de todos los componentes necesarios para su funcionamiento, además, la hoy conocida por todos Intel lanzaba el primer microprocesador. A día de hoy estos microprocesadores contienen miles de millones de transistores, lo que ha permitido que: por una parte, un ordenador personal de pequeño tamaño y bajo coste sea capaz de realizar tareas de cierto coste computacional que hace algunas décadas parecían imposibles y, por otra parte, que los superordenadores más grandes del mundo, formados por miles de estos microprocesadores, sean capaces de realizar simulaciones científicas muy costosas que nos ayudan en la comprensión del mundo que nos rodea.

La computación de altas prestaciones[1], o HPC por sus siglas en inglés, hace referencia al conjunto de técnicas de programación, herramientas y equipos capaces de proporcionar rendimientos en términos de potencia de cálculo muy elevados y que se utilizan para resolver problemas muy complejos que requieran de un gran coste computacional. Algunas aplicaciones del HPC son la química computacional, el análisis genómico, la ingeniería asistida por ordenador, la predicción del tiempo, el modelado de riesgo financiero, la simulación de armas nucleares, el análisis de datos o el entrenamiento de modelos de inteligencia artificial. Todas estas aplicaciones son muy costosas y para poder realizarlas se hace uso de la computación paralela, que permite resolver problemas cada vez más grandes en tiempos razonables, apoyada tanto en *hardware* como en *software* especialmente diseñado y optimizado para esas aplicaciones, y que se comentarán a continuación.

[1]Término intercambiable con el de supercomputación, menos usado actualmente.

	SYSTEM	SPECS	SITE	COUNTRY	CORES	RMAX PFLOP/S	POWER MW
1	**Summit**	IBM POWER9 (22C, 3.07GHz), NVIDIA Volta GV100 (80C), Dual-Rail Mellanox EDR Infiniband	DOE/SC/ORNL	USA	2,414,592	148.6	11.4
2	**Sierra**	IBM POWER9 (22C, 3.1GHz), NVIDIA Tesla V100 (80C), Dual-Rail Mellanox EDR Infiniband	DOE/NNSA/LLNL	USA	1,572,480	94.6	7.44
3	**Sunway TaihuLight**	Shenwei SW26010 (260C, 1.45 GHz) Custom Interconnect	NSCC in Wuxi	China	10,649,600	93.0	15.4
4	**Tianhe-2A (Milkyway-2A)**	Intel Ivy Bridge (12C, 2.2 GHz) & TH Express-2, Matrix-2000	NSCC Guangzhou	China	4,981,760	61.4	18.5
5	**Frontera**	Dell C6420, Xeon Platinum 8280 28C 2.7GHz, Mellanox InfiniBand HDR	TACC/U of Texas	USA	448,448	23.5	-

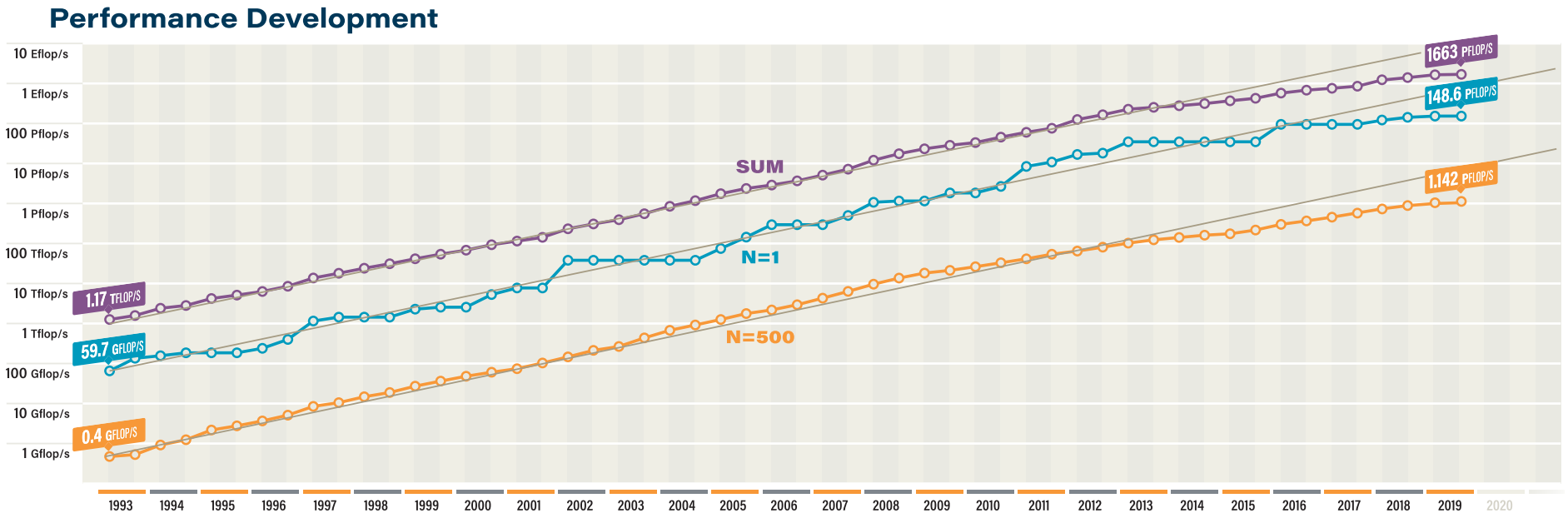


Figura 4.1: Top 5 superordenadores y evolución histórica de la lista Top500.

4.1. *Hardware*

El *hardware* es la parte física de un ordenador referida a los componentes electrónicos y mecánicos presentes en él. Es la parte que más ha evolucionado a lo largo de los años, principalmente en el número de transistores de los procesadores pero también en la cantidad de memoria, las redes de conexión y últimamente las tarjetas gráficas. Estas últimas, desgraciadamente, no son tan útiles para aplicaciones donde haya una gran frecuencia de lectura y escritura a disco, como es el caso.

4.1.1. Potencia y Top500

Como es de esperar, el objetivo y deseo de los usuarios de la computación de altas prestaciones es el de tener acceso a la mayor cantidad de potencia computacional posible para utilizar en sus programas. La potencia computacional referida a la capacidad de realizar cálculos se suele medir en FLOPs (Operaciones de coma flotante por segundo) mediante el estándar LINPACK, una librería de resolución de sistemas de ecuaciones lineales. Dos veces al año se publica el famoso *ranking* del Top500 [22], con los 500 superordenadores más potentes del planeta. Un pequeño póster-resumen de la lista de noviembre de 2019 puede verse en el Anexo A, siendo la información más relevante la presente en la Figura 4.1.

En la parte superior de la misma se puede observar como el superordenador más potente de ese momento era el *Summit*, con cerca de dos millones y medio de núcleos y una potencia o rendimiento máximo de 148 PFLOPs (10^{15} FLOPs). En la gráfica de la evolución histórica se muestran tres series de datos, la superior (SUM) representa la suma de la potencia de los 500 ordenadores recogidos en la lista en cada año, la intermedia (N=1) muestra la potencia del superordenador más potente de la lista en cada año y la última (N=500) la potencia del último superordenador de la lista. Esta gráfica es importante por dos motivos:

El primero es que es fácil ver la progresión histórica en la potencia de cálculo a lo largo de los años, donde se puede observar, por ejemplo, que el ordenador más potente de la lista de 2019 iguala a la suma de los 500 más potentes de la lista de 2011, o que la suma de los 500 más potentes de 2019 ya supera la exaescala (10^{18} FLOPs). Como curiosidad, debido a la situación de confinamiento que la pandemia de la COVID-19 ha supuesto a fecha de redacción de este proyecto, la iniciativa de computación distribuida de plegado de proteínas Folding@Home[2], en ayuda para luchar contra dicha enfermedad, se convirtió en abril de 2020 en el sistema con mayor potencia de cálculo del mundo, con cerca de 2,5 EFLOPs, mayor que la suma de los 500 mayores superordenadores en ese momento, gracias a los sistemas de sus usuarios repartidos por todo el planeta, entre los cuales me encuentro.

El segundo motivo es que si se observan con detalle las líneas sólidas se puede apreciar que la tendencia (pendiente de las curvas, nótese que el eje y está representado en escala logarítmica) que se venía manteniendo a lo largo de los años empieza a desaparecer, presentando una ralentización en los últimos años. Esto se relaciona con la famosa Ley de Moore, que establecía que el número de transistores en un microprocesador se doblaba

[2]Precedida históricamente por otras como la iniciativa de búsqueda de vida extraterrestre SETI@Home, que en este 2020, y tras muchos años, ha dejado de distribuir tareas.

cada dos años. El problema es que el tamaño de los transistores actuales más avanzados es muy pequeño, de unos 7 nm, lo que empieza a suponer problemas a la hora de fabricarlos y, si se sigue reduciendo su tamaño, habrá que tener en cuenta posibles efectos cuánticos. Esta reducción del tamaño de los transistores es muy beneficiosa, principalmente por motivos de disipación de calor y eficiencia por consumir menos energía por unidad de potencia de cálculo. Esto supondrá que la dificultad para hacer un procesador más potente aumente, y con ello los costes de diseño y fabricación, siendo necesario un cambio de filosofía y apostar por la utilización de más procesadores menos potentes en vez de uno solo de más potencia, intentando que el consumo energético no se dispare.

4.1.2. Procesadores

El componente más importante de un ordenador es, sin duda, el procesador o CPU (Unidad de procesamiento central), que se encarga de gestionar el funcionamiento del sistema mediante la realización de operaciones aritméticas, lógicas y de entrada/salida. Un aspecto a destacar es que el concepto de procesador como tal ha ido evolucionando con los años debido a las limitaciones continuas de aumentar su potencia, como puede ser el aumento en la frecuencia de reloj. Las CPUs modernas (el chip en sí) cuentan con varios núcleos o *cores*, y estos, a su vez, con varios hilos o *threads* (normalmente dos). A nivel físico son los núcleos, llamados también procesadores lógicos, los que permiten la ejecución paralela de procesos de manera independiente aunque, mediante *software*, es posible simular que cada uno de los hilos de estos núcleos es una unidad lógica en sí misma.

Los procesadores, como se comentaba anteriormente, han avanzado mucho, principalmente en potencia pero también en eficiencia. Los procesadores punteros actualmente cuentan con miles de millones de transistores, lo que ha sido posible gracias a la reducción del tamaño de estos, y decenas de núcleos. Por ejemplo, el AMD EPYC ROME, uno de los más potentes en la actualidad junto a los Intel XEON, cuenta con aproximadamente $4 \cdot 10^{10}$ transistores de 7 nm, 64 núcleos y 128 hilos de ejecución, todo ello en un único chip no apto para todos los bolsillos, pues ronda los 7000 euros.

4.1.3. Jerarquía de memoria y cachés

Uno de los factores más importante en HPC es el acceso a los datos en memoria ya que puede limitar de manera considerable el rendimiento de la aplicación. En problemas científicos y de ingeniería es muy común que haya un bucle principal de alto coste computacional y temporal, en el que grandes cantidades de datos entran y salen constantemente de la memoria principal (RAM) al procesador en el que se realizan operaciones con ellos. Esta entrada y salida de datos puede que sea lenta y, dependiendo de la aplicación específica, que sea el factor limitante de la velocidad de ejecución de la misma ya que el procesador puede realizar las operaciones a mayor velocidad que la velocidad de transferencia de estos datos. Otro factor limitante puede ser la baja velocidad de escritura a un disco duro de almacenamiento, aunque esto es menos común hoy en día.

Como se puede observar en la Figura 4.2 hay varios tipos de memoria presentes en cualquier ordenador común. Por un lado, en la cima de la pirámide, encontramos los registros del procesador, muy rápidos pero muy pequeños en capacidad. Por otro lado, en la base, encontramos los discos duros de almacenamiento magnéticos clásicos, con mucha capaci-

dad pero lentos. La parte especialmente interesante para las aplicaciones científicas es la intermedia, donde se encuentran las memorias caché y la memoria RAM.

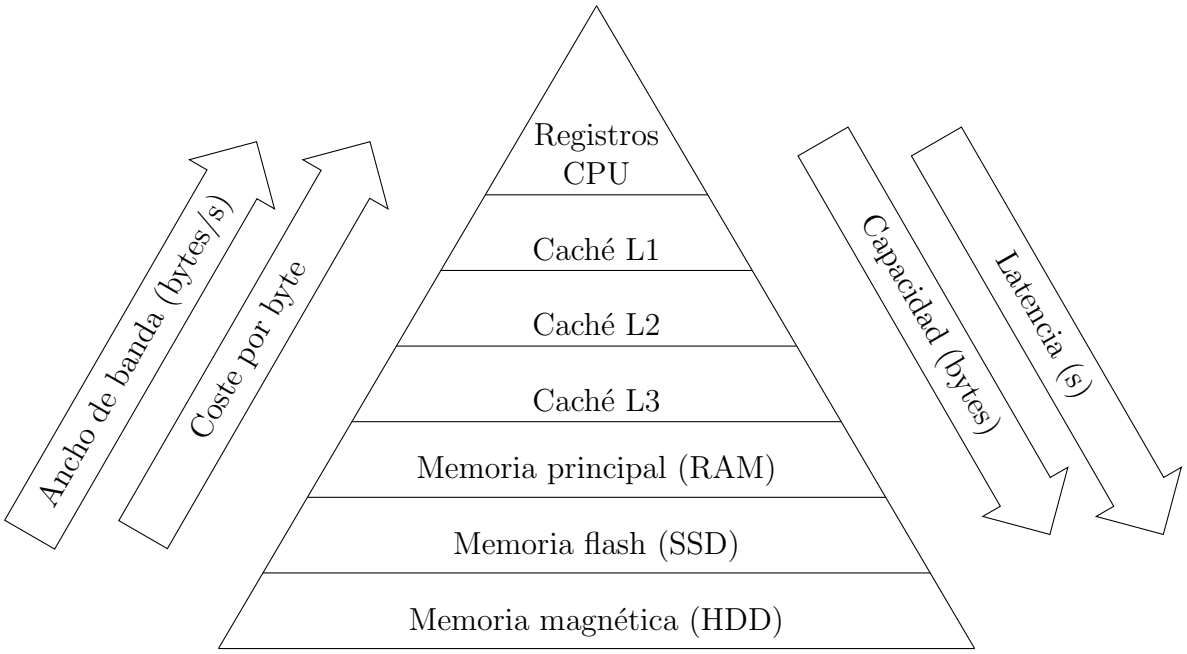


Figura 4.2: Jerarquía de memoria y sus características.

La memoria RAM es importante en los casos en los que el dominio a simular sea muy grande, ya que todos los datos necesarios para ejecutar el programa deben ser almacenados en esta memoria y puede que no quepan. Como se verá más adelante, una de las optimizaciones realizadas al código ha sido la de cambiarlo a precisión doble, lo que supone que la memoria RAM necesaria será el doble de la original, llegando a valores considerables que solo se pueden encontrar en superordenadores.

La memoria caché es uno de los parámetros importantes de diseño de los procesadores modernos, y es que cuanto más memoria caché L1 disponga el chip más caro será. La memoria caché está estructurada en tres niveles (L1, L2, L3), cada uno de ellos más rápido, caro y de menor capacidad. La memoria L3 es compartida a todos los núcleos del mismo procesador. La L2 puede ser compartida o única a cada núcleo. Por último, la caché L1 es única a cada uno de los núcleos y es la más importante porque es la más rápida después de los registros del procesador. Estas memorias caché almacenan la información que es más probable que la aplicación necesite a continuación (líneas de caché) para que el proceso de cargar estos datos sea más rápido. Si los datos que se necesitan no están cargados en caché el procesador los tendrá que buscar en la RAM, siendo este proceso mucho más lento. Cuanto más grandes sean estas líneas de caché (en términos de bytes) más posibilidades hay de que los datos deseados se encuentren en ellas.

Este concepto está muy presente en los desarrolladores experimentados que diseñan su código para intentar optimizar el uso del acceso a caché. Un ejemplo de esto se verá en el capítulo siguiente de la mano de la operación de transponer una matriz con una técnica llamada caché *blocking*.

4.1.4. Tipos de sistemas

La clasificación estándar de los tipos de arquitecturas de ordenadores viene dado por la taxonomía de Flynn, que muestra la relación entre el número de instrucciones concurrentes y el flujo de datos disponible en relación con el número de procesadores, entendidos como unidades de cálculo y no como el chip completo tal y como veíamos en la sección anterior. Son cuatro las posibilidades:

- **SISD** (Una instrucción, un dato): Ordenador secuencial (no paralelo), basado en la arquitectura de Von-Neumann, con un único procesador lógico y una memoria[3]. Se ejecuta una única instrucción en cada ciclo de reloj. Actualmente obsoleto.
- **SIMD** (Una instrucción, múltiples datos): Ordenador vectorial, con varios procesadores lógicos pero una única unidad de control. Se ejecuta en paralelo la misma instrucción pero con datos diferentes. Entran aquí la mayoría de móviles y ordenadores personales.
- **MISD** (Múltiples instrucciones, un dato): Ordenador redundante, muy poco utilizado, únicamente en aplicaciones donde el fallo es crítico, como las espaciales.
- **MIMD** (Múltiples instrucciones, múltiples datos): Ordenador paralelo con varios procesadores autónomos independientes que realizan simultáneamente instrucciones sobre datos diferentes. Entran aquí los sistemas distribuidos.

Según lo visto, la arquitectura utilizada en los sistemas de altas prestaciones es la MIMD, que a su vez está formada por componentes SIMD, por lo que se hace necesario para el correcto entendimiento de la materia profundizar un poco más recurriendo a una subclasificación de estos sistemas, donde se tiene en cuenta el acceso a la memoria de los diferentes procesadores. Así, se pueden distinguir dos tipos, los de memoria compartida y los de memoria distribuida.

Ordenadores de memoria compartida

Son aquellos sistemas en los que todos los núcleos de uno o varios procesadores comparten la memoria, lo que permite que cualquiera de ellos pueda acceder a cualquier posición de memoria a través de la red interna de conexión de la placa base. Dentro de esta clasificación encontramos los sistemas UMA (Acceso uniforme a memoria), en los que los diferentes núcleos de uno o varios procesadores tienen acceso a una única memoria del sistema, solo una. Por otro lado encontramos los sistemas NUMA (Acceso no uniforme a memoria), que guardan cierta similitud con los sistemas de memoria distribuida, y en los que cada procesador tiene su propia memoria física estando todas ellas conectadas mediante una red de conexión interna. En este último caso la velocidad de acceso a la memoria dependerá de si se accede a la memoria local o a una remota. Un pequeño esquema del sistema NUMA puede encontrarse en la Figura 4.3, adaptada de Hager [23], donde encontramos en un mismo nodo dos procesadores de cuatro núcleos conectados a su propia memoria y entre ellos, además de las memorias caché que antes se comentaban. Para paralelizar instrucciones en este tipo de sistemas se utiliza el estándar OpenMP (*Open Multi-Processing*) [24], que se detallará más adelante.

[3]Por memoria se entiende la principal del sistema, conocida comúnmente como la memoria RAM, aunque sería la de disco en ausencia de la primera.

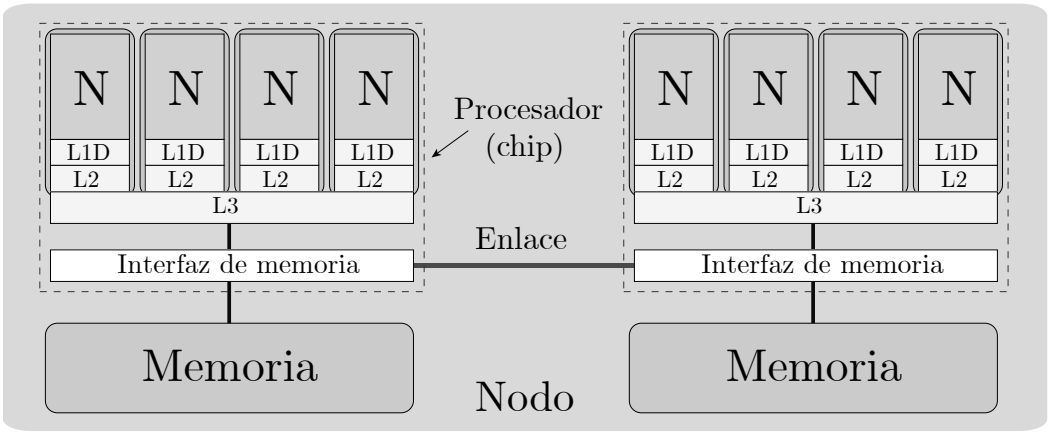


Figura 4.3: Esquema de un sistema NUMA.

Ordenadores de memoria distribuida

Son aquellos sistemas en los que un procesador está exclusivamente conectado a su propia memoria local, sin que ningún otro pueda acceder a ella de forma directa. Para poder acceder a la memoria de otros procesadores se utiliza lo que se conoce como paso de mensajes o comunicaciones, que consiste en enviar o recibir paquetes de información de los distintos procesadores. Estos mensajes se transportan mediante una red de conexión de alta velocidad, lo que se verá en la siguiente subsección. Para paralelizar este tipo de sistemas se utiliza el estándar MPI (*Message Passing Interface*) [25].

Así pues, se puede concluir que los superordenadores modernos son sistemas de memoria distribuida formados por un conjunto de nodos conectados por una red de conexión de alta velocidad, donde cada nodo es un sistema con memoria local compartida formado normalmente por dos o más procesadores (chips) en una misma placa base que, a su vez, están formados por varios núcleos y varios hilos. De nuevo, un pequeño esquema de estos sistemas se puede observar en la Figura 4.4, adaptada de Hager [23], donde varios nodos NUMA con dos procesadores cada uno se conectan entre ellos mediante una red de comunicaciones. La paralelización de estos sistemas, según se ha visto, se realizará mediante una combinación híbrida de OpenMP y MPI.

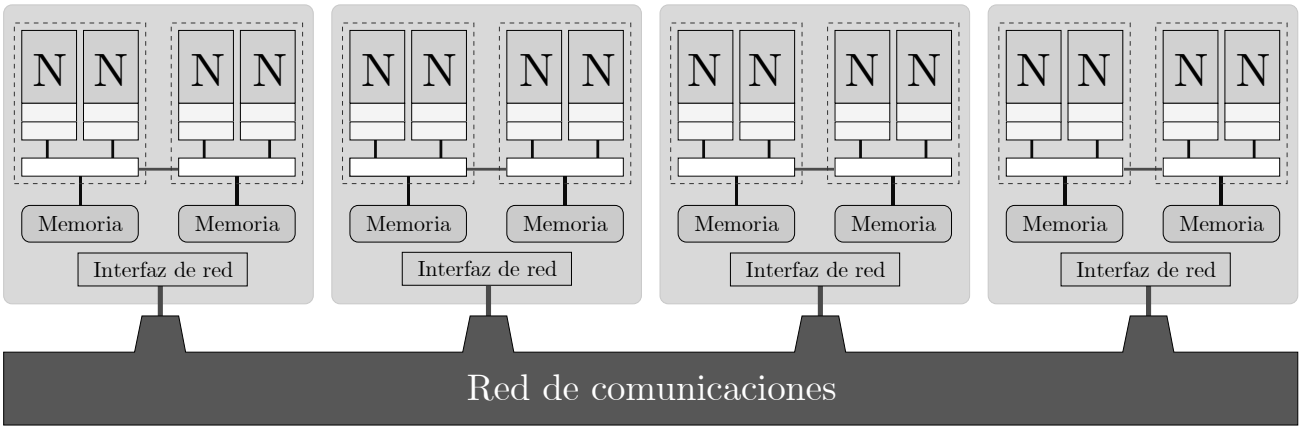


Figura 4.4: Esquema de la arquitectura de un superordenador.

4.1.5. Redes de conexión

Tal y como se acaba de comentar, los superordenadores modernos utilizan redes de conexión de alta velocidad para poder entablar comunicaciones entre nodos y así poder transferir información entre ellos. Si volvemos de nuevo a la Figura 4.1, vemos como en las especificaciones de los superordenadores *Summit, Sierra* y *Frontera* aparece algo como *Mellanox Infiniband*, que es precisamente la red de conexión que usan. La tecnología *Infiniband* es un tipo de conexión de alta velocidad y baja latencia que está presente en un gran número de superordenadores, llegando a velocidades de transferencia de hasta 400 Gbits/s y compitiendo con interfaces como la *Ethernet*. En estos casos, tanto los cables, *switches* como conectores los proporciona la empresa israelí Mellanox, adquirida por NVIDIA por 6900 millones de dólares. Si volvemos a la Figura 4.2, su rendimiento estaría entre la memoria RAM y los discos SSD en cuestiones de ancho de banda y latencia.

En términos de las redes de conexión hay dos características importantes a tener en cuenta al elegir el tipo de red. La primera es la latencia, que representa el tiempo que se tarda en iniciar la conexión entre dos procesadores. La otra es el ancho de banda, que representa la velocidad de transferencia a la que se intercambian los datos. Como es obvio, cuanto menor sea la latencia y mayor el ancho de banda más rápido funcionará un programa en un sistema de memoria distribuida. Sin embargo, conocer las especificaciones puede suponer una diferencia en el rendimiento de un programa, por ejemplo, enviar un mensaje largo suele ser más rápido que enviar muchos mensajes cortos aunque la cantidad total de datos transmitida sea la misma debido a la latencia de la red.

4.1.6. MareNostrum y SuperMUC-NG

Para la correcta realización del proyecto no era suficiente con la utilización de ordenadores personales ya que se necesitaba un gran número de procesadores para algunas de las tareas a realizar. Gracias al acceso privilegiado del tutor de este proyecto ha sido posible acceder a dos superordenadores, el MareNostrum y el SuperMUC-NG en el que, dependiendo de su disponibilididad[4], se han podido ejecutar los códigos.

El superordenador MareNostrum está localizado en el Centro de Supercomputación de Barcelona (BSC) y es conocido por muchos debido a que se encuentra en una ubicación poco común pero asombrosa, dentro de una capilla (ver Figura 4.5). El MareNostrum, en su última versión que es la IV[5], es el superordenador más potente en el territorio español, con una potencia máxima de 11 PFLOPs, 400 TB de memoria RAM y 153000 núcleos de procesamiento, lo que le hace ocupar el lugar número 30 en la lista del Top500 de Noviembre de 2019.

El superordenador SuperMUC-NG está localizado en el Centro Leibniz de Supercomputación (LRZ) en Alemania y es el segundo más potente a nivel europeo y el noveno a nivel mundial según el Top500 antes mencionado, ya que cuenta con 300000 núcleos de procesamiento y 700 TB de memoria RAM, llegando a una potencia máxima de 27 PFLOPs. Una de sus particularidades es que utiliza refrigeración líquida en vez de por aire.

[4] A finales de mayo de 2020 SuperMUC fue hackeado para minar criptomonedas, por lo que se restringió su acceso.

[5] Aunque ya se encuentra en construcción la versión V, que se espera que se estrene a principios del año 2021 y cuente con una potencia de unos 200 PFLOPs.

Figura 4.5: Ubicación del MareNostrum IV.

4.2. *Software*

El *software* es la parte no física de un ordenador formada por un conjunto de instrucciones que los componentes físicos ejecutan para que este funcione, incluyendo el sistema operativo, programas, librerías, etc. La evolución con los años de esta parte ha estado ligada principalmente a la mejora y aparición de nuevos algoritmos que realizan de manera más efectiva las tareas propuestas y que, en ocasiones, han sido posibles únicamente por el rápido desarrollo del *hardware*. A continuación se presenta una pequeña explicación del *software* utilizado para el desarrollo de este proyecto.

4.2.1. Linux

Prácticamente la totalidad del Top500 de superordenadores usan Linux o alguna de sus distribuciones como sistema operativo. Esto es debido a varias razones: el hecho de que sea un software libre, es decir, modificable y distribuíble; su funcionalidad, dado que hay un gran número de librerías científicas ya desarrolladas bajo este sistema; su rendimiento, con muy pocos procesos en segundo plano y, por acabar la lista, su facilidad de administración, con una gran cantidad de usuarios conectados al mismo tiempo, colas de espera para ejecutar los programas, permisos y directorios de cada usuario, etc.

4.2.2. FORTRAN

El FORTRAN (*FORmula TRANslating system*) es un lenguaje de programación de alto nivel (sintaxis adecuada al ser humano) utilizado casi en su totalidad para cálculo numérico y científico. Hay varios motivos por lo que esto es así. En primer lugar, porque es un lenguaje compilado, no interpretado, por lo que se traduce la sintaxis escrita por un humano a lenguaje máquina antes de la ejecución, aumentando así su velocidad al ser

posible activar optimizadores que, según la arquitectura de cada máquina, transforman las instrucciones de manera que se adecúen a esta para que la ejecución del código sea más rápida. Esto presenta también algunos inconvenientes, como que es más difícil de depurar. En segundo lugar, porque al ser un lenguaje antiguo[6] contiene una gran cantidad de librerías de cálculo científico listas para utilizar en la aplicación del usuario, entre ellas las que permiten paralelizar el código.

Al ser un lenguaje compilado, como su nombre indica, será necesario compilar el programa desarrollado antes de ejecutarlo. Hay varias alternativas para elegir compilador, siendo los más conocidos el compilador de Intel y el gratuito gfortran. Los compiladores tienen un gran efecto en la velocidad con la que posteriormente se ejecutará el programa. Además, al compilar se pueden añadir ciertas opciones, conocidas como *flags*, para elegir el tipo de optimización, opciones para depurar, etc. Es por esto que los superordenadores con procesadores de Intel son en la actualidad la gran mayoría, como se puede ver en el Anexo A, dado a que su compilador conoce mejor la arquitectura de los propios procesadores y es capaz de realizar mejores optimizaciones del programa a compilar. Un ejemplo de compilación de un programa vía terminal puede observarse en el Código 4.1, con el compilador de Intel y una *flag* de optimización.

Código 4.1: Ejemplo de compilación de un programa por consola.

```
1 # Compila el programa hello.F90, creando el ejecutable hello.exe
2 $ ifort -O3 -o hello.exe hello.F90
3 # Ejecuta el programa
4 $ ./hello.exe
```

4.2.3. OpenMP

Como se ha comentado anteriormente, los procesadores actuales tienen más de un núcleo lógico y a su vez varios hilos, pero estos no siempre pueden ser aprovechados por los programas que se ejecutan, y es que estos deben ser adaptados por el desarrollador para que sean capaces de funcionar en paralelo. La paralelización de código en sistemas de memoria compartida, donde cualquier unidad lógica puede acceder a cualquier posición de memoria, es más sencilla que en los sistemas de memoria distribuida. Para ello se utiliza el estándar OpenMP, una API que contiene una serie de directrices a llamar desde el código que al compilar introduce las instrucciones para que el ejecutable final lo haga de forma paralela. Su funcionamiento es relativamente sencillo y es que al comienzo de la ejecución del programa este se ejecuta únicamente en uno de los hilos del procesador (*master thread*) de manera secuencial y es el desarrollador el que indica cuales son las regiones que deben ejecutarse en paralelo tal y como se puede observar en la Figura 4.6, adaptada de Hager [23]. La paralelización, independientemente del tipo de sistema, se puede implementar a nivel funcional, donde cada unidad de procesamiento ejecuta instrucciones diferentes, o a nivel datos, donde todas las unidades de procesamiento ejecutan las mismas instrucciones pero con datos diferentes. Un ejemplo de este último caso son las regiones con bucles, donde códigos de ámbito científico gastan la mayoría de su tiempo de ejecución, y en los que la variable o índice a iterar se reparte entre los hilos disponibles.

[6]La primera versión, de 1957, ejecutaba programas escritos en tarjetas de cartón perforadas.

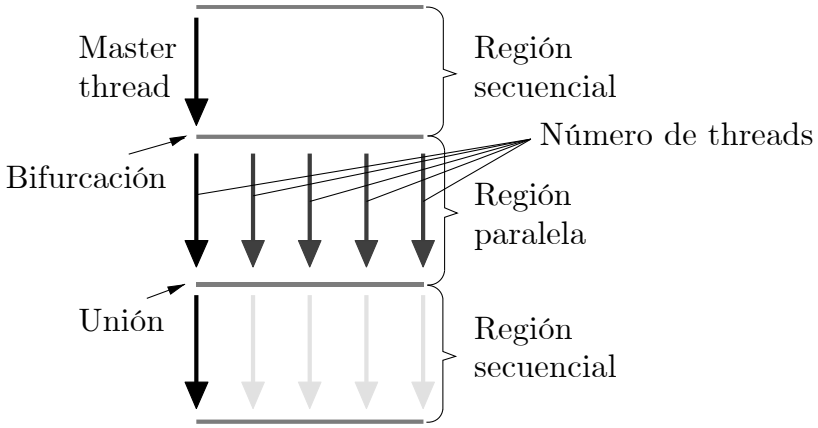


Figura 4.6: Estructura de ejecución paralela con OpenMP.

Al implementar con OpenMP, dado el carácter de compartir la memoria, hay que tener cuidado en que varios núcleos o hilos no escriban en la misma posición de memoria de manera simultánea, pues puede producir problemas. Para ello se pueden declarar ciertas variables globales o privadas a cada unidad lógica, aunque la memoria física sea compartida. También existen directivas de sincronización, en caso de que haya que esperar a que todos los procesos hayan realizado alguna operación y muchas funcionalidades más, por lo que es muy versátil. El Código 4.3 muestra un ejemplo básico del célebre *hello_world* en paralelo con OpenMP, donde se muestran algunas funcionalidades. El número de hilos con los que se ejecuta la aplicación puede ser definido antes de la ejecución, mediante la variable de entorno *OMP_NUM_THREADS*, o dentro del propio programa de manera dinámica mediante la subrutina *OMP_SET_NUM_THREADS*[7]. Para compilar un programa con soporte para OpenMP es necesario compilarlo con la *flag* -qopenmp para el compilador de Intel y -fopenmp para gfortran. Así, un ejemplo de compilación de un programa por consola definiendo previamente a la ejecución el número de hilos deseados para ejecutarlo puede encontrarse en el Código 4.2.

Código 4.2: Ejemplo de compilación y ejecución de un programa con OpenMP.

```
1  # Seleccionar número de threads deseados
2  $ export OMP_NUM_THREADS = 8
3  # Compila el programa
4  $ ifort -O3 -o -qopenmp hello_omp.exe hello_omp.F90
5  # Ejecuta el programa
6  $ ./hello_omp.exe
```

Código 4.3: Ejemplo de hello_world paralelo con OpenMP.

```
1  program hello_omp
2  use omp_lib !API OpenMP
3  implicit none
4
5  integer :: maxth, numth, thid !Declaración de variables, th significa threads
6
7  !Esta región solo la ejecuta el master thread
8  maxth = OMP_GET_MAX_THREADS() !Consulta el número máximo de hilos disponibles en el sistema
9  numth = OMP_GET_NUM_THREADS() !Consulta el número de hilos con los que se ha ejecutado el programa
10
```

[7]El hecho de representar las rutinas en mayúsculas es para diferenciar las provistas por APIs como OpenMP y MPI de las creadas por el usuario.

```
11 call OMP_SET_NUM_THREADS(maxth) !Establece el número de threads al máximo del sistema
12
13 !$OMP PARALLEL  !Empieza la región paralela
14       thid = OMP_GET_THREAD_NUM() !Consulta el número de identificación de cada thread
15       write(*,*), 'Hola desde el thread: ', thid !Escribe por pantalla desde cada thread
16 !$OMP END PARALLEL  !Finaliza la región paralela
17 end program
```

4.2.4. MPI

Para el caso de los sistemas de memoria distribuida, donde cada procesador solo tiene memoria local, se utilizan mensajes para intercambiar datos. Para ello se utiliza el estándar MPI, donde varias copias del mismo programa se ejecutan de forma paralela. Su utilidad reside en que hay ciertas directivas que permiten asignar acciones o datos diferentes a cada uno de los procesos MPI. A diferencia de la paralelización con OpenMP, con MPI las tareas se asignan a procesos virtuales del sistema y no directamente a los hilos del procesador. De esta manera, uno puede ejecutar tantos procesos MPI paralelos como desee, incluso más de uno para cada núcleo. Como es obvio, es imprescindible saber las especificaciones de la máquina para ejecutar el número óptimo de procesos, de lo contrario el sistema sufriría una ralentización importante por sobrecarga. El Código 4.5 muestra un ejemplo básico de *hello_world* en paralelo con MPI, donde se muestran algunas funcionalidades. El número de procesos con los que se ejecutará el programa se define antes de ejecutar la aplicación, en el mismo comando de ejecución, como se puede ver en el Código 4.4.

Código 4.4: Ejemplo de compilación y ejecución de un programa con MPI.

```
1 # Compila el programa
2 $ mpiifort -O3 -o hello_mpi.exe hello_mpi.F90
3 # Ejecuta el programa con el número de procesos deseado
4 $ mpirun -np 8 ./hello_mpi.exe
```

Código 4.5: Ejemplo de hello_world paralelo con MPI.

```
1  program hello_mpi
2  use mpi !API MPI
3  implicit none
4
5  integer :: nummpi, myid, ierror
6
7  call MPI_INIT(ierror)  !Inicialización MPI
8  call MPI_COMM_SIZE(MPI_COMM_WORLD, nummpi, ierror)  !Consulta el número de procesos MPI
9  call MPI_COMM_RANK(MPI_COMM_WORLD, myid, ierror) !Consulta el número de identificación de cada ↩
       ↪ proceso
10
11 write(*,*), 'Hola desde el proceso: ', myid !Escribe por pantalla desde cada proceso
12
13 call MPI_FINALIZE(ierror) !Finaliza MPI
14 end program
```

Las ejecuciones más comunes de los programas finales suelen ser: o mandar un proceso MPI a cada nodo de la red del superordenador y en ese nodo usar OpenMP con sus núcleos/hilos o lanzar tantos MPIs como núcleos tenga un nodo y a su vez OpenMP con los hilos. Un ejemplo de esta paralelización híbrida se puede observar en la Figura 4.7, donde para cada proceso MPI (en este caso tres) se utiliza OpenMP con el número de hilos disponibles (en este caso 5). A parte de códigos propios desarrollados, programas de

simulación comerciales como STARCCM+ o ANSYS también utilizan estas tecnologías por debajo de la interfaz de usuario para que este se pueda centrar en los aspectos físicos de la simulación y no tanto en los computacionales al ejecutarlo en servidores de cálculo.

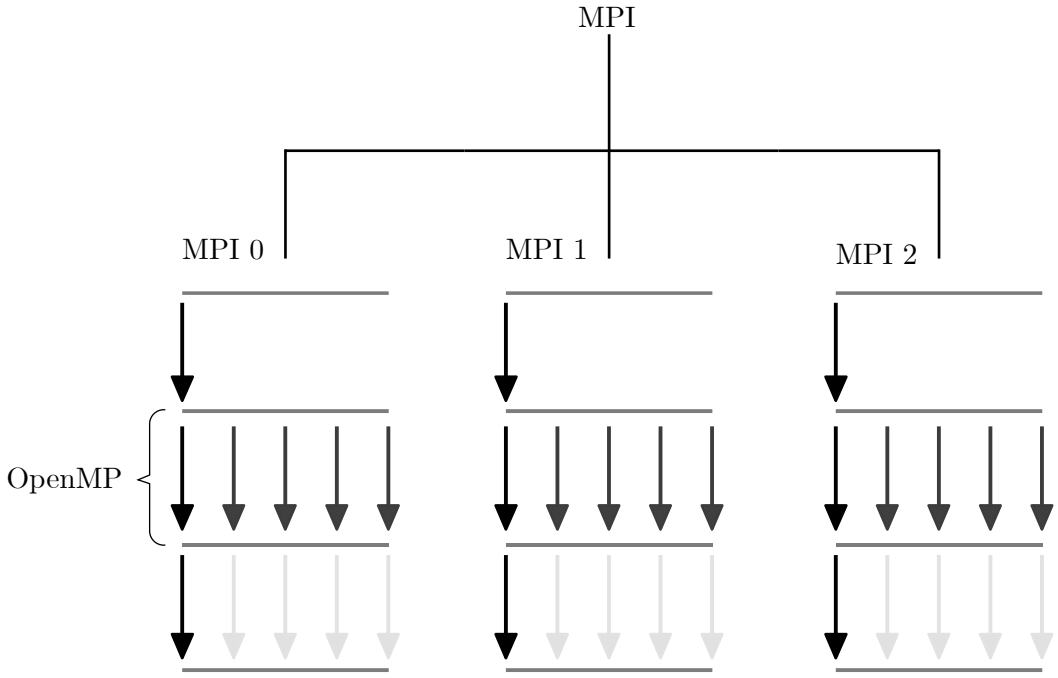


Figura 4.7: Estructura de ejecución paralela híbrida con OpenMP y MPI.

Un aspecto muy importante en este tipo de sistemas distribuidos al utilizar MPI son las comunicaciones entre procesadores con las que se transmiten los mensajes (datos) unos a otros y que se verán en detalle en el siguiente capítulo, pues son una de las partes más importantes del desarrollo de este trabajo.

4.2.5. FFTW3

Como se comentaba en el Capítulo 3, el código utiliza métodos espectrales basados en la transformada rápida de Fourier (FFT). Para ello se utiliza la librería de software libre FFTW3 [26], que es considerada una de las más rápidas en realizar las transformadas discretas de Fourier y que cuenta con múltiples opciones como pueden ser transformadas en varias dimensiones y de tamaño arbitrario, manejo de datos tanto reales como complejos o funciones adaptadas para esquemas paralelos. Además, su documentación es excelente y su uso muy sencillo.

4.2.6. HDF5

La cantidad de datos generados por este tipo de simulaciones es muy alta, y estos deben ser almacenados de manera ordenada y segura para poder acceder a ellos tantas veces como sea necesario. Por tener un orden de magnitud, la simulación de $Re_\tau = 10000$ [3] ha producido una base de datos de cerca de 250 TB. El procesamiento de estos datos es, al igual que la propia simulación, muy costosa en tiempo. De hecho, según Jiménez [11], el tiempo que se tarda en pos-procesar los datos generados suele ser el doble de lo que se tarda en simular por lo que algunos lo hacen en la propia ejecución del código.

Escribir o leer tal cantidad de datos no es algo trivial, y se hace necesario utilizar un formato de archivo que permita una fácil organización. En este caso, al igual que en otros campos científicos, se utilizan archivos en formato HDF5 [27], un sistema jerárquico (por capas) de almacenamiento ilimitado de datos que permite, entre otros, almacenar datos complejos de varias dimensiones, compartirlos de forma sencilla en un solo archivo, rapidez de lectura/escritura o soporte para multitud de lenguajes y plataformas. Además, la librería incluye una capa superior a la del compilador deseado para la aplicación (Intel o gfortran) y que lo reemplaza por ser superior ya que tiene en cuenta la localización de los archivos necesarios y las funciones de dicha librería utilizadas en el código de manera que no haya problemas al ejecutarlas fundamentalmente de manera paralela.

4.3. Indicadores de rendimiento

La finalidad de usar sistemas de HPC no es otra que la de poder simular problemas más grandes más rápido. Para simular sistemas más grandes es necesario más memoria RAM, que es sencillamente la suma de las memorias de los nodos individuales. Sin embargo, para ejecutar problemas más rápido y elegir la cantidad óptima de procesadores con los que hacerlo no es trivial, y es que se hace necesario evaluar el rendimiento del algoritmo final, quizás con un dominio menor, para ver como este evoluciona con el número de procesadores utilizado. Para caracterizar este rendimiento se suelen utilizar dos métricas, el *speedup* o aceleración y la eficiencia, que se calcularán al evaluar los resultados.

La métrica más importante es la aceleración (4.1), que se define como el cociente entre el tiempo de ejecución con un número de procesadores de referencia s y con un número de procesadores p mayor a este.

$$S(p) = \frac{t_s}{t_p} \tag{4.1}$$

En casos rutinarios se suele definir $s = 1$ pero para casos grandes con acceso a superordenadores este número lo define el propio usuario. Tomando el caso base idealizado, si se dispone de p procesadores se esperaría finalizar el trabajo en una fracción $1/p$ del tiempo inicial (una aceleración máxima de p). Sin embargo, algunos aspectos como tiempos de espera en comunicaciones hacen que esto no sea posible. Es importante destacar también que en casi todos los algoritmos hay partes que se ejecutan de manera secuencial sin importar el número de procesadores disponibles y cuyo rendimiento no va a mejorar con ellos, siendo ejemplos de esto los sobrecostes al inicializar regiones paralelas. Esto es precisamente lo que establece la llamada Ley de Amdahl: el incremento de velocidad de un programa ejecutado en paralelo está limitado por la fracción secuencial del mismo.

Los límites en la aceleración vienen dados por el hecho de que cuantos más procesadores se utilicen el problema se divide en más trozos y es posible que estos no tengan el coste computacional suficiente como para que su procesador asociado funcione de manera óptima. Una manera de caracterizar esto es con la métrica de la eficiencia (4.2), que caracteriza la desviación con respecto a la aceleración ideal, cuyos valores varían entre 0 y 1, siendo 1 el hecho de aprovechar todos los procesadores de forma óptima y viceversa.

$$E(p) = \frac{S(p)}{p} \tag{4.2}$$

Capítulo 5

Aspectos computacionales

Prácticamente cualquier código de simulación científica a gran escala deberá ser ejecutado en un superordenador durante un tiempo, y es por eso que los aspectos computacionales del mismo son de gran importancia, ya que estos pueden ser capaces de reducir ese tiempo de ejecución en una cantidad considerable. Por ello, el desarrollador del código no debe centrarse únicamente en los aspectos físicos y numéricos del problema a simular sino que debe prestar atención también a su implementación, sobre todo si el tiempo de simulación es alto, del orden de meses.

Para el problema concreto del DNS que simula el código LISO hay varios aspectos a tener en cuenta para una ejecución óptima del mismo, siendo algunos de ellos capaces de sufrir alguna optimización adicional pese a la especial atención y dedicación a la que ha sido sometido durante tantos años de desarrollo, pues las primeras versiones datan de hace más de 15 años, y que se comentarán a continuación.

5.1. División del dominio

Uno de los aspectos claves de este tipo de problemas que requieren mucha capacidad de cálculo es que el dominio suele ser de dimensiones considerables. Es por eso que, gracias a la computación paralela, se puede dividir el dominio entre los procesadores disponibles tanto para acelerar los cálculos como para que sea posible el almacenamiento en memoria de las variables, pues dominios muy grandes no siempre podrían ser almacenados en un único sistema. Por ejemplo, en códigos de esta naturaleza pueden llegar a estar presentes hasta 10 variables del tamaño del dominio, que ya de por sí es extenso.

Hay diversas estrategias a seguir para dividir un dominio computacional entre los procesadores disponibles, dependiendo de la aplicación puede que una división sea mejor que otra. La estrategia utilizada en el código es la de dividir el dominio, que recordemos era un canal que se podía modelar como un paralelepípedo, en varios planos según el eje x, de manera que cada procesador tenga un número de planos. Cabe destacar que el código es capaz de ejecutarse únicamente cuando el número de planos (puntos en x) es divisible por el número de procesadores, de manera que cada procesador tenga varios planos en su versión normal y un solo plano en la versión final, que es la que se ejecuta en algún superordenador con tantos procesadores como puntos en x del dominio que se desee simular. En la Figura 5.1 se puede observar como sería esta división con N procesadores.

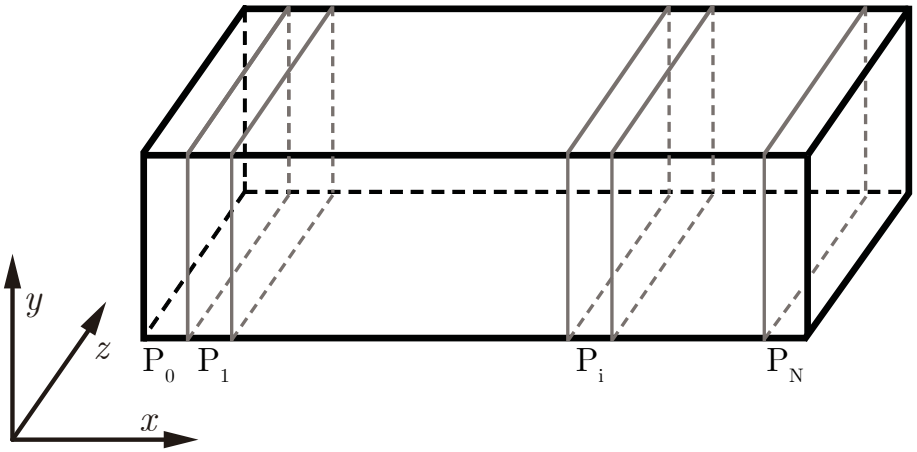


Figura 5.1: División del dominio entre los procesadores.

Esta división del dominio es una de las tareas que primero se realizan en el código, de manera que cada procesador sepa el número de planos que le corresponden y su posición respecto a la dimensión total. Como se comentaba en el Capítulo 3, para ser capaces de realizar la transformada rápida de Fourier (FFT) en una de las direcciones es necesario tener los datos de esa dimensión almacenados en la misma memoria local e idealmente de forma contigua para aumentar la velocidad de ejecución. Recordemos que las FFT eran necesarias para cambiar del dominio de Fourier al dominio físico y así poder calcular la helicidad de forma más rápida. Así, para poder ejecutar la FFT en dirección z no hay ningún problema[1], pues todos los datos en esa dimensión están en la memoria local a cada procesador. Sin embargo, cuando se deba ejecutar la FFT en dirección x se puede observar que no sería posible, pues los procesadores no tienen todos los datos en esa dirección. Es en este punto cuando entra la segunda forma de paralelización, las líneas. Mediante comunicaciones, las cuales se desarrollarán en la siguiente subsección, se realiza lo que se conoce como una transpuesta global donde los procesadores intercambian datos de manera que se pase de tener un número de planos yz en dirección x a únicamente un número de líneas en dirección x, donde las líneas totales son el número de elementos en uno de los planos yz, es decir, el número de elementos en y multiplicado por el número de elementos en z. Para conocer el número de líneas que recibe cada procesador se divide el número de líneas totales entre el número de procesadores de manera que no es necesario tener un plano completo sino únicamente un fragmento. En la Figura 5.2 se puede observar este cambio de planos a líneas.

Tras este cambio en el almacenamiento local de los datos para realizar las operaciones pertinentes es necesario volver al estado anterior y así poder empezar una nueva iteración temporal, de manera que se realiza el proceso inverso tanto de transpuestas como de comunicaciones. Antes de pasar a desarrollar estas dos operaciones, y dado que ya se ha presentado de forma general el funcionamiento del algoritmo, se sintetiza de forma esquemática la secuencia que este sigue, abstrayéndose del significado físico presente en la resolución, y centrándose en las operaciones de ámbito computacional a realizar:

[1]El único problema en este caso, como veremos más adelante, es que los datos no están contiguos en memoria, por lo que una transpuesta local de esos datos será necesaria para que lo estén.

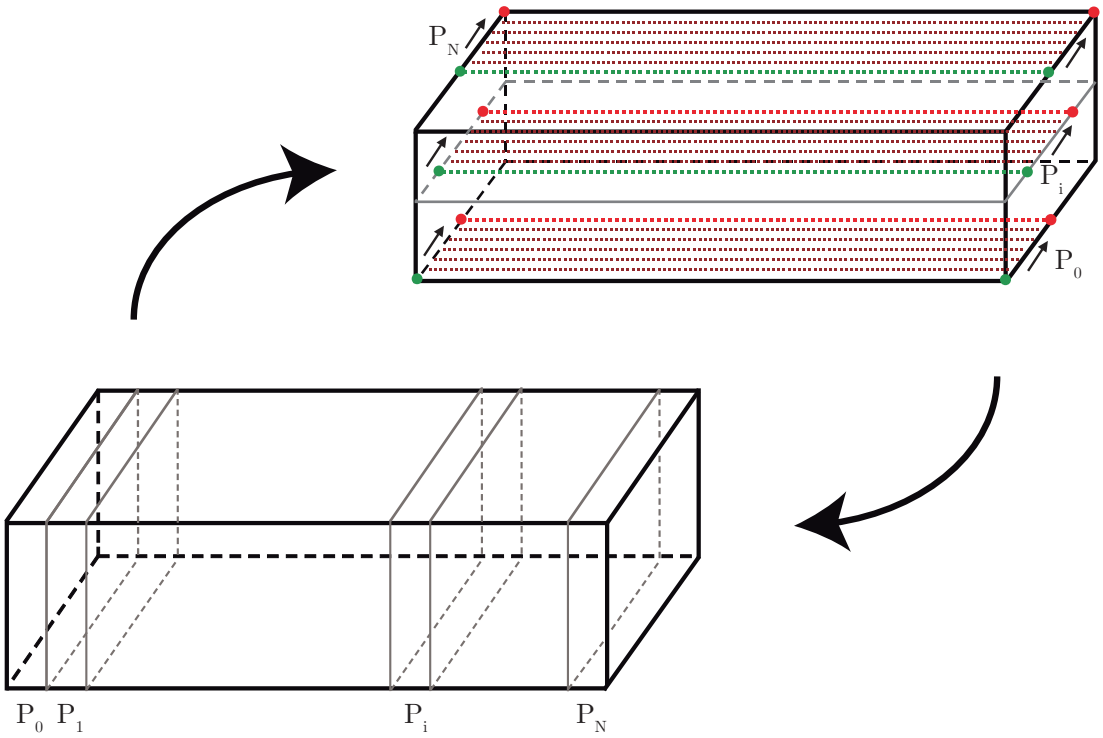


Figura 5.2: Estrategia de paralelización en planos-rectas.

Estructura computacional del algoritmo

1. Datos ordenados en planos yz en dirección x.
2. Transpuesta local de datos ordenados según yz a zy.
3. Función que opera en dirección z. (FFT)
4. Transpuesta global de planos zy a líneas en x.
5. Función que opera en dirección x. (FFT)
6. Transpuesta global de líneas en x a planos zy.
7. Función que opera en dirección z. (FFT)
8. Transpuesta local de datos ordenados según zy a yz.

Esta secuencia de operaciones es necesario realizarla para cada una de las variables que se pretenden resolver, en este caso las tres velocidades y las tres vorticidades, ya que cada una de ellas es diferente, por lo que la secuencia se repite seis veces. En el siguiente capítulo se verá como algunas de las optimizaciones intentan mejorar la velocidad del algoritmo cambiando el orden de las operaciones o mediante ligeras modificaciones.

5.2. Comunicaciones

Las comunicaciones o envío de mensajes se utilizan para intercambiar datos entre los procesadores en sistemas de memoria distribuida, y son un aspecto importante ya que son costosas en términos de tiempo, cosa que se cuantificará en el capítulo siguiente. Algunas

de las características a tener en cuenta son qué proceso envía el mensaje y cuál lo recibe, cuál es el puntero en memoria desde el que se envía y recibe, qué tipo de dato se está mandando, cuál es el tamaño de estos datos, etc. El estándar MPI cuenta con bastantes subrutinas que permiten realizar estas comunicaciones, y las hay de varios tipos. Dentro de una clasificación general se encuentran las comunicaciones punto a punto y las comunicaciones globales.

Las comunicaciones punto a punto intercambian un mensaje entre dos procesos MPI, siendo uno el que envía y el otro el que recibe. Ejemplos de este tipo de comunicaciones son *MPI_SEND* o *MPI_RECV* que, como su nombre indica, se utilizan para enviar y recibir mensajes. Por otro lado encontramos las comunicaciones globales, en las que participan todos los procesos MPI presentes, de manera que se pueden realizar operaciones diferentes. Algunos ejemplos son *MPI_BCAST*, donde un proceso envía un mensaje idéntico a todos los demás, o *MPI_ALLTOALL* que envía datos desde todos los procesos a todos los procesos.

La otra clasificación que se puede hacer es la de diferenciar las comunicaciones que bloquean (*blocking*) y las que no (*non-blocking*). Las comunicaciones *blocking* no finalizan hasta que todos los datos hayan sido enviados y recibidos por completo. Esto tiene ventajas e inconvenientes, como ventaja destaca la posibilidad de reutilizar el *buffer* desde el que se mandaban los datos, así ahorrando memoria. La desventaja es que si el proceso de destino no está listo para recibir el que envía está parado sin realizar trabajo. Ejemplos comunes son los vistos anteriormente *MPI_SEND* o *MPI_RECV*. En cuanto a las comunicaciones *non-blocking*, estas te permiten enviar todos los datos y seguir realizando trabajo sin tener que esperar a que el destinatario reciba el mensaje. Un aspecto a tener en cuenta es que, dependiendo del tipo de programa a desarrollar, se debe comprobar que la comunicación ha finalizado (mediante *MPI_WAIT*) antes de poder utilizar el *buffer* o variable donde se reciben los datos, de lo contrario el programa daría error.

En el caso del código LISO este está implementado en dos versiones, una con comunicaciones punto a punto y otra con comunicaciones globales, que se detallarán a continuación. En el siguiente capítulo se presentarán las propuestas de optimización del código, entre las que se encuentra la implementación de comunicaciones *non-blocking*.

5.2.1. MPI_SENDRECV

Una de las implementaciones (la principal) respecto a las comunicaciones es la que utiliza la función *MPI_SENDRECV* [25], cuya estructura se puede observar en el Código 5.1. Esta función es una combinación de *MPI_SEND* y *MPI_RECV*, con la misma funcionalidad, por lo que es una comunicación de tipo *blocking* punto a punto entre dos de los procesos. Recordemos que para que el funcionamiento sea óptimo ambos procesos, el que envía y el que recibe, deben estar preparados para la comunicación. Es por eso que la implementación óptima será esa en la que los procesos estén el menor tiempo posible en espera, es decir bloqueados. Para intercambiar datos entre todos los procesos se utiliza una estructura de comunicaciones en forma de hipercubo de desarrollo propio, no disponible en las APIs, donde las comunicaciones se hacen por parejas de forma ordenada para que ninguno de ellos esté en espera durante toda la secuencia de comunicaciones. Este algoritmo es óptimo para un número de procesadores igual a la potencia enésima de 2 (2^n) y se implementa

mediante el operador lógico de *OR exclusivo* bit a bit entre el número de procesos y el identificador de cada proceso. Un ejemplo de este tipo de estructura con 8 procesos (2^3) puede observarse en la Figura 5.3, donde cada uno de los cubos representa una iteración de comunicaciones, siendo el conjunto de todos ellos el resultado final en el que todos los procesos se han comunicado con todos los procesos.

Código 5.1: Estructura de la función MPI_SENDRECV.

```
1  call MPI_SENDRECV(sendbuf, !Puntero inicial del buffer de envío
2                    sendcount, !Numero de elementos a enviar
3                    sendtype, !Tipo de los datos a enviar
4                    dest, !Número de identificación del destinatario
5                    sendtag, !Etiqueta del mensaje a enviar
6                    recvbuf, !Puntero inicial del buffer de recepción
7                    recvcount, !Número de elementos a recibir
8                    recvtype, !Tipo de datos a recibir
9                    source, !Número de identificación del remitente
10                   recvtag, !Etiqueta del mensaje a recibir
11                   status, !Variable de estados de la operación de recibir (finalizada o no)
12                   MPI_COMM_WORLD, !Comunicador global
13                   ierror) !Índice de error
```

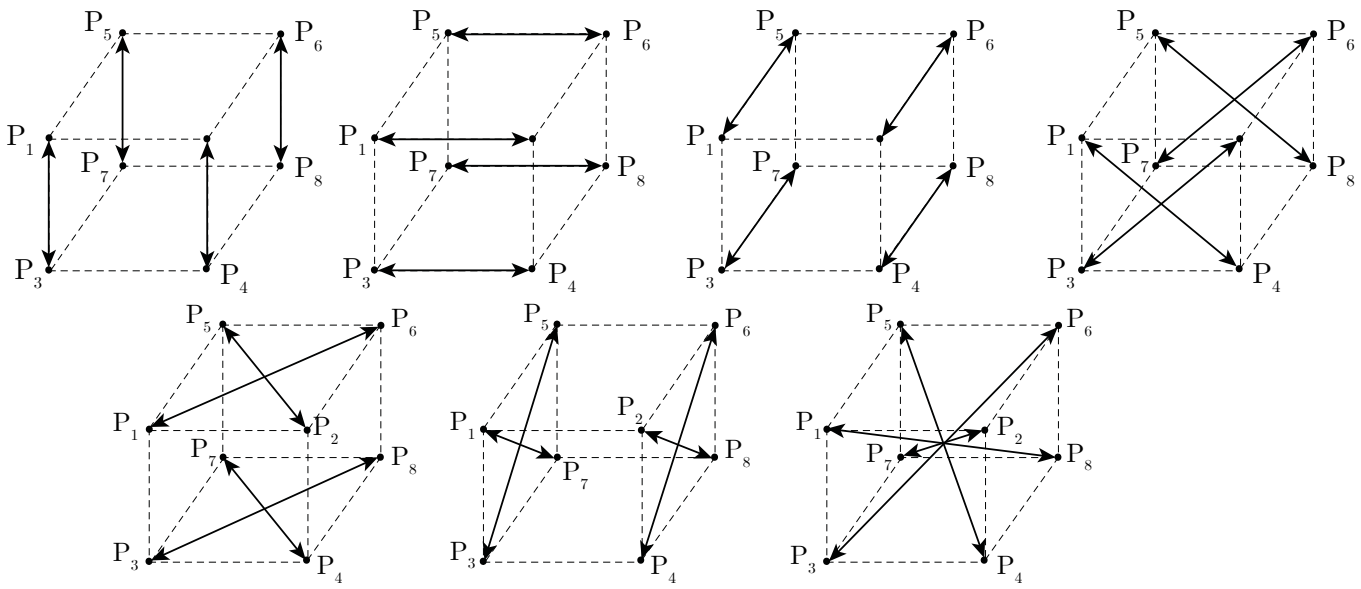


Figura 5.3: Estructura de comunicaciones hipercubo para 8 procesos.

5.2.2. MPI_ALLTOALL

Otra de las implementaciones utiliza la función *MPI_ALLTOALL*, que permite comunicaciones colectivas y que se puede entender como una transposición de datos entre procesos, de manera que el proceso i envía el dato o grupo de datos en la posición j al proceso j que lo guarda en la posición i, como se puede ver en la Figura 5.4. Recordemos que tanto a esta función como a la de la sección anterior la llaman todos los procesos, cuya memoria es privada bien de forma física o de forma virtual, de manera que cada uno envía a todos y recibe de todos. El mayor problema de esta rutina es que si el programa se ejecuta con un elevado número de procesos es posible que la red de conexión se sature y las comunicaciones sean extremadamente lentas, por lo que es necesario tener conocimientos de la

arquitectura de la máquina donde se va a ejecutar.

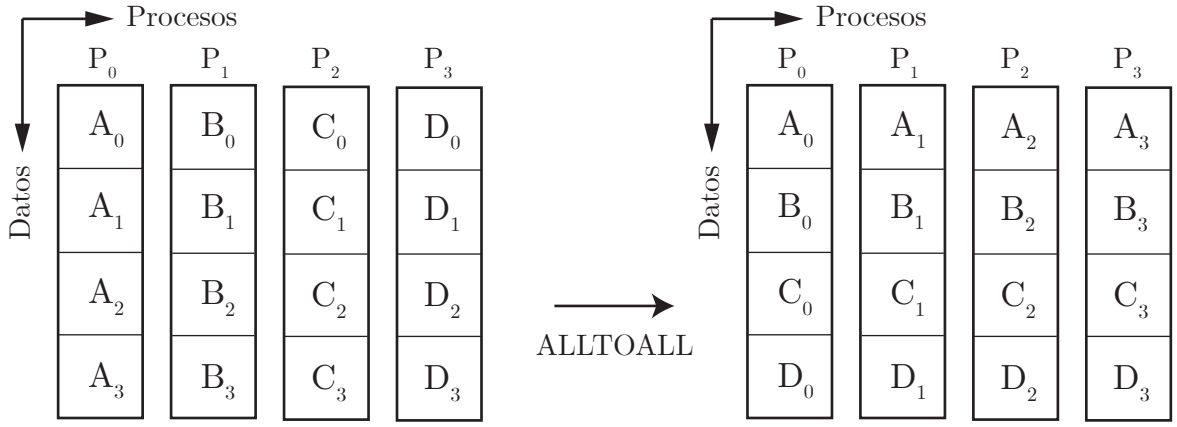


Figura 5.4: Representación gráfica de *MPI_ALLTOALL*.

La función original solo se puede utilizar cuando hay tantos grupos de datos del mismo tamaño como número de procesos ejecutándose. Si se quiere enviar un número de datos distinto a cada proceso es necesario utilizar una versión modificada de la función original *MPI_ALLTOALLv*, cuya estructura puede observarse en el Código 5.2 junto con una imagen aclaratoria, la Figura 5.5.

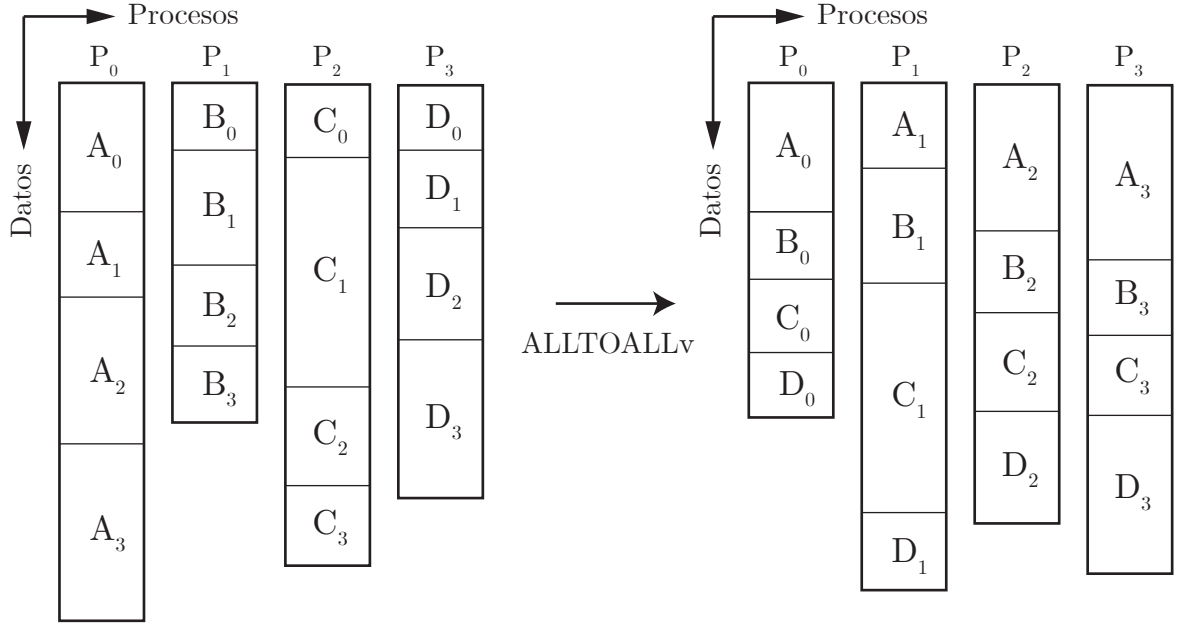


Figura 5.5: Representación gráfica de *MPI_ALLTOALLv*.

Código 5.2: Estructura de la función MPI_ALLTOALLv.

```
1  call MPI_ALLTOALLv(sendbuf, !Puntero inicial del buffer de envío
2                     sendcounts, !Lista de elementos a enviar a cada proceso
3                     sdispls, !Lista con el desplazamiento respecto del puntero de envío inicial desde el que ↩
                          ↪ tomar los datos de salida a cada proceso
4                     sendtype, !Tipo de los datos a enviar
5                     recvbuf, !Puntero inicial del buffer de recepción
6                     recvcounts, !Lista de elementos a recibir de cada proceso
7                     rdispls, !Lista con el desplazamiento respecto del puntero de recepción inicial desde el ↩
                          ↪ que colocar los datos de entrada de cada proceso
8                     recvtype, !Tipo de datos a recibir
9                     MPI_COMM_WORLD, !Comunicador global
10                    ierror) !Índice de error
```

5.3. Transpuesta de una matriz

Una de las operaciones que se repite un gran número de veces en el código LISO son las transpuestas, pero no tanto para realizar cálculos si no para que las comunicaciones se realicen de forma correcta. La operación de transponer una matriz bidimensional es muy simple, cambiar filas por columnas y viceversa en una nueva matriz, lo que se puede observar en la Figura 5.6, adaptada de Hager [23], y en el Código 5.3. Para elementos de más dimensiones será necesario establecer el orden que se desea después de realizar la operación. En operaciones como la transpuesta de una matriz bidimensional cuadrada de lado n aparecen dos bucles anidados cada uno de longitud n por lo que el coste de la operación es del orden de $\mathcal{O}(n^2)$. Pese a ser una operación muy fácil de comprender e implementar, hay dos conceptos computacionales importantes que intervienen en ella y que pueden mejorar su rendimiento.

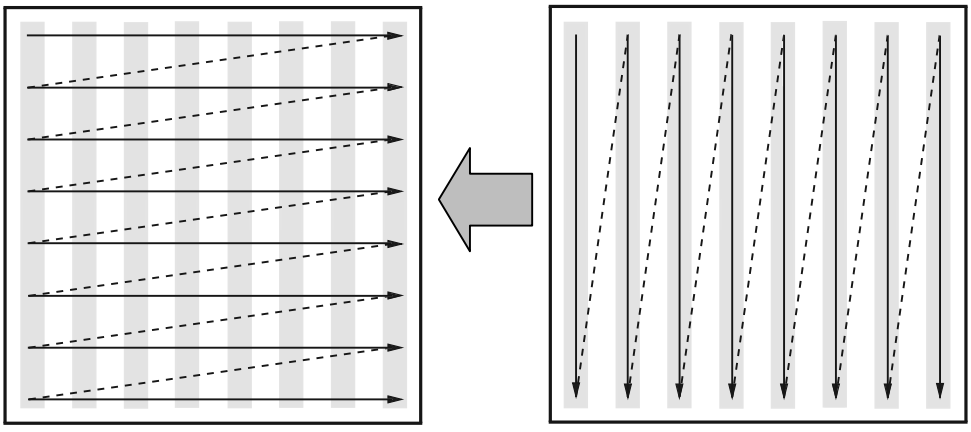


Figura 5.6: Esquemática de la transpuesta de una matriz.

El primer concepto clave es el orden de almacenamiento de los datos, ya que en los códigos científicos abundan los datos multidimensionales. Si recordamos, la memoria en la que estos se guardan se puede visualizar como una tira o línea continua y es necesario relacionar las posiciones de esa tira continua con las de una matriz multidimensional con la que los humanos nos sentimos más cómodos utilizando, aunque desde el punto de vista del lenguaje máquina es exactamente lo mismo. Cada lenguaje de programación utiliza un tipo de organización para los datos dimensionales. En la Figura 5.7, adaptada de Hager [23], se pueden observar los dos tipos de orden de almacenamiento para una matriz de datos

bidimensional. En la imagen izquierda el índice que varía rápido (posiciones contiguas en memoria) corresponde a las columnas de una misma fila, siendo el lenguaje C un ejemplo de este orden. Por el contrario, en la imagen de la derecha observamos como el índice rápido es el de las filas dentro de una misma columna, que es el orden utilizado por FORTRAN.

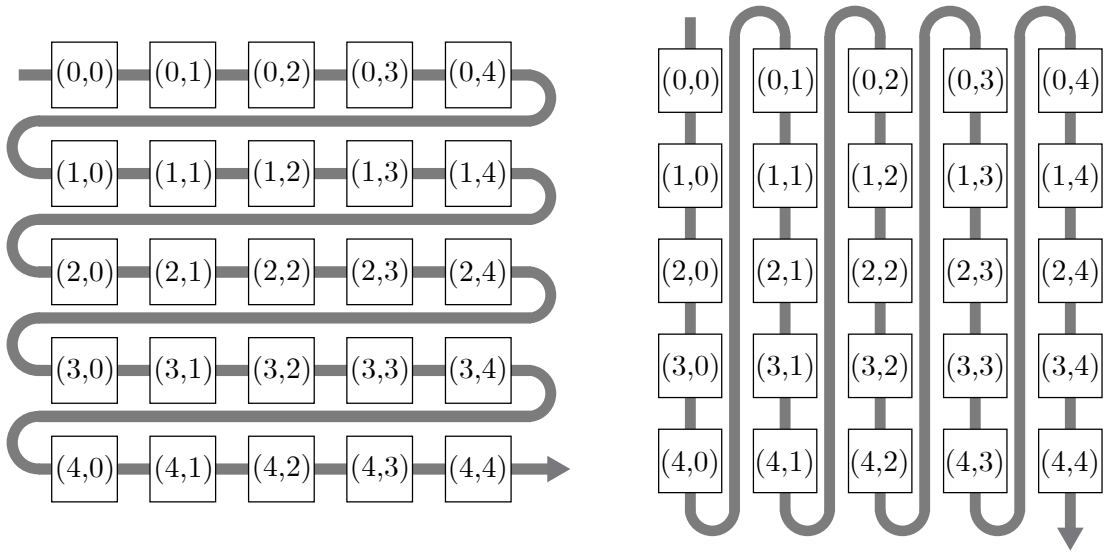


Figura 5.7: Órdenes de almacenamiento en memoria.

Saber el orden de almacenamiento, aparte de para desarrollar correctamente la aplicación, podría *a priori* afectar al rendimiento de la misma en ciertas operaciones en la que no se aprovechen los datos cargados en caché. Además, suele ser común que se tarde más en escribir a memoria que en que se lea de ella. Para el caso que nos interesa, el de las matrices transpuestas, se ha realizado una prueba con matrices bidimensionales cuadradas de hasta una dimensión de 10000 elementos por lado para observar si este cambio de índices (que varíe primero en el bucle interior el índice rápido al leer o al escribir) tiene algún efecto en el rendimiento de la operación. Los Códigos 5.3 y 5.4 muestran estas operaciones. Como se puede observar en la Figura 5.10 ambos casos son prácticamente idénticos en rendimiento, por lo que se considera que no tiene un efecto considerable. Más adelante se explicará dicha figura en detalle.

Código 5.3: Transpuesta de una matriz, índice rápido escribe.

```
1 do j = 1,N
2       do i = 1,N
3             B(i,j) = A(j,i) !Lee de A y escribe en B
4       enddo
5 enddo
```

Código 5.4: Transpuesta de una matriz, índice rápido lee.

```
1 do j = 1,N
2       do i = 1,N
3             B(j,i) = A(i,j)
4       enddo
5 enddo
```

El segundo concepto clave, y que se ha desarrollado de manera introductoria en el capítulo anterior, es el de la memoria caché. Si el acceso a los datos necesarios por el programa en un momento dado están cargados en las líneas de caché este será rápido, de lo contrario no lo será por lo que es importante que el desarrollador cuide estos detalles al escribir el código, siendo otras operaciones de interés las multiplicaciones de vectores o matrices. Para la aplicación de la matriz transpuesta existe una técnica para nada trivial llamada caché *blocking* (no confundir con el de comunicaciones). Esta técnica consiste en engañar al compilador haciéndole creer que la matriz sobre la que se va a operar tiene unas dimensiones menores a las reales. Para ello se define un factor de bloqueo (*bloque*) para uno de los bucles (*blocking* simple) o para los dos bucles anidados (*blocking* doble). Este factor de bloqueo varia según los componentes de cada máquina por lo que se suelen hacer unas pruebas preliminares (*benchmarks*) antes de la ejecución final de la aplicación midiendo varios valores para este factor de bloqueo que busca utilizar una mayor cantidad de los datos guardados en caché que la versión normal, ganando así algo de velocidad. Es importante destacar que el número de operaciones totales es el mismo en ambos casos y únicamente se modifica la secuencia de acceso a memoria. Ejemplos del caché *blocking* simple y doble pueden observarse tanto en las Figuras 5.8 y 5.9 como en los Códigos 5.5 y 5.6.

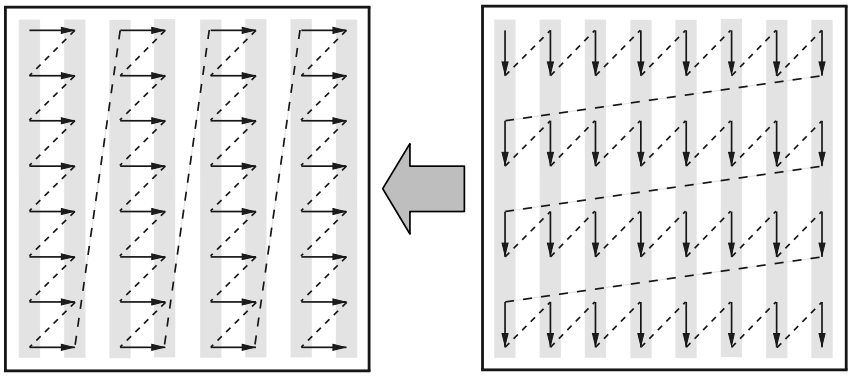


Figura 5.8: Esquemática de la transpuesta de una matriz con caché blocking simple.

Código 5.5: Transpuesta de una matriz con caché blocking simple.

```
1 do ii = 1,N,bloque !Bucle de 1 hasta N con paso bloque
2       do j = 1,N
3             do i = ii,ii+bloque-1
4                   B(i,j) = A(j,i)
5             enddo
6       enddo
7 enddo
```

Código 5.6: Transpuesta de una matriz con caché blocking doble.

```
1 do jj = 1,N,bloque
2       do ii = 1,N,bloque
3             do j = jj,jj+bloque-1
4                   do i = ii,ii+bloque-1
5                         B(i,j) = A(j,i)
6                   enddo
7             enddo
8       enddo
9 enddo
```

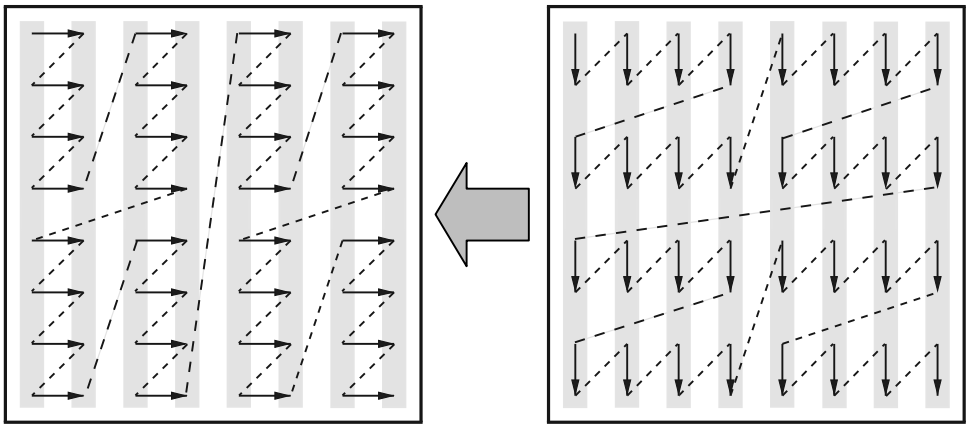


Figura 5.9: Esquemática de la transpuesta de una matriz con caché blocking doble.

Si volvemos a observar la Figura 5.10, en escala logarítmica en ambos ejes, se puede observar como la diferencia entre *blocking* simple y doble es también prácticamente despreciable, aunque se hará un estudio en el siguiente capítulo. Sin embargo, se observa como el rendimiento que ofrece introducir un factor de bloqueo es sustancial, llegando a medio orden de magnitud aproximadamente. En esta figura no es importante el tiempo total de la operación, pues depende de la máquina en la que se ejecute, si no en la diferencia de las tendencias entre las diferentes aproximaciones para ejecutar dicha operación. Se observa que alrededor de un tamaño de 1500 elementos por lado la diferencia entre las versiones normales y las que usan caché *blocking* empieza a ser notable. Por último, también se observa que las pendientes de todos los métodos corresponden con la de n^2 tal y como se comentaba al principio de la sección.

5.4. Gestión de memoria

Otro de los aspectos clave en este tipo de códigos es la gestión de memoria. Asignar de manera dinámica elementos a variables existentes (ir aumentando su tamaño conforme avanza el código) es muy ineficiente en velocidad al tener que reasignar el espacio que estas variables ocupan en memoria. Para subsanar este problema se pre-asignan todas las variables de tamaño considerable al inicio del código, haciendo así que el sistema reserve toda esa cantidad de memoria de una vez y evitando modificar su tamaño durante la ejecución del código. Esta asignación se realiza mediante módulos (similares a las librerías) que tienen la ventaja de que pueden ser usados en subrutinas de manera que cuando se le pasen los argumentos de entrada, si es una variable con la memoria ya asignada, se utilice el mismo espacio ya reservado en memoria, no asignando uno nuevo a una variable temporal con la que se trabaja en esa rutina y después se elimina.

Un ejemplo de este uso son las matrices transpuestas que antes se detallaban. Al realizar la operación el número de elementos total es el mismo, cambiando solo su posición. En vez de tener dos variables del mismo tamaño en bytes, una para la matriz original y otra para la transpuesta, se utiliza una variable temporal de menos tamaño para ir reasignando las nuevas posiciones de los elementos en la matriz original, utilizando la memoria ya reservada en vez de reservar una nueva.

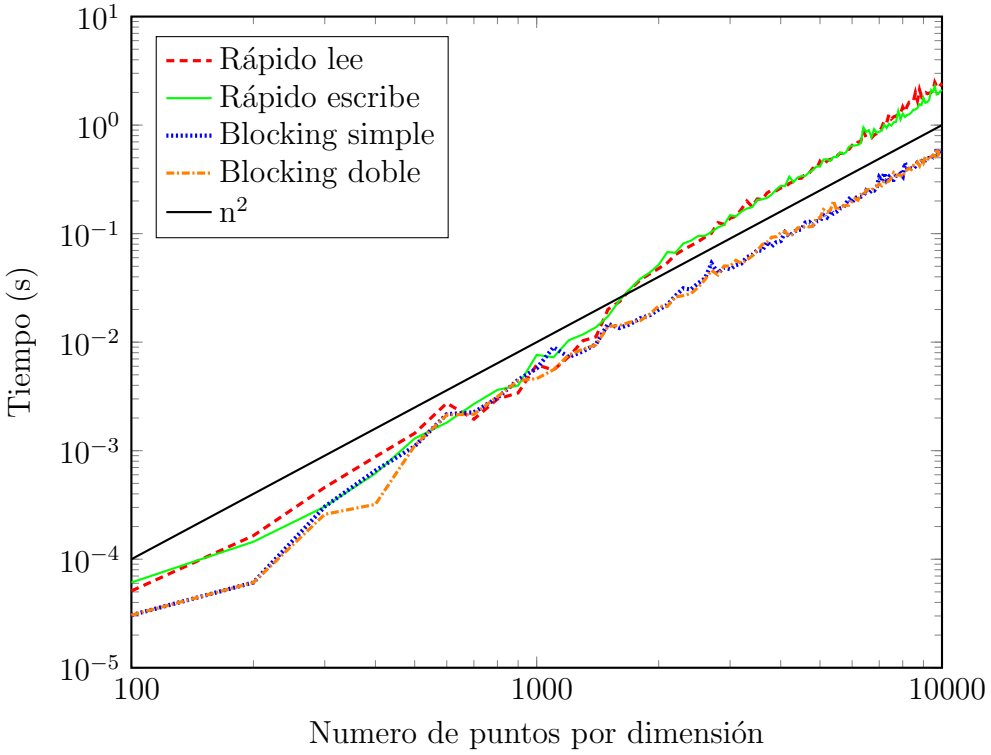


Figura 5.10: Comparación de formas de transponer una matriz.

Capítulo 6

Optimizaciones y resultados

El objetivo principal de este trabajo es el de optimizar el código de dos maneras distintas. La primera optimización ha sido la de cambiar el código de simple precisión a doble precisión y, la segunda, la de proponer varias alternativas para el problema de las comunicaciones, buscando un tiempo de ejecución menor. Ambas optimizaciones serán explicadas en detalle a continuación y, además, los resultados obtenidos serán analizados para evaluar de forma cuantitativa las mejoras realizadas, si las hubiera.

La optimización de códigos de simulación es una práctica común en los campos científicos, ya sean códigos propios del personal investigador como códigos comerciales de resolución de problemas, principalmente en el campo ingenieril con programas de simulación de multifísicas. Estas optimizaciones no son sencillas ya que los desarrolladores originales seguramente tuvieron en cuenta muchos de los aspectos computacionales que pueden afectar al rendimiento en la ejecución. Sin embargo, las arquitecturas de los ordenadores, tal y como hemos visto en capítulos anteriores, evolucionan rápidamente con los años y surgen nuevas técnicas para hacer más rápida la ejecución de estos programas.

Pequeños cambios en las estructuras de los algoritmos pueden suponer un ahorro considerable, tanto en tiempo como en dinero, en aquellos casos en los que la simulación es muy grande y está pensada para ejecutarse durante varios meses en superordenadores. Este es por ejemplo el caso de las simulaciones DNS como las que realiza el código LISO. Una de las características a tener en cuenta es que el rendimiento de los códigos de simulación depende en gran medida de la máquina específica donde se va a ejecutar, por lo que normalmente no hay una solución ideal y universal sino que antes de la ejecución final es necesario hacer unas pruebas de *benchmarking* para corroborar qué procedimientos funcionan mejor en esa máquina. Precisamente por esto se ha desarrollado también un programa que ejecuta todas las versiones de las rutinas de comunicaciones hechas, de manera que nos permita saber qué tipo de comunicación o estructura de algoritmo presenta un mayor rendimiento en la máquina deseada a fin de adaptarse a ella.

El coste de unidades de procesamiento en superordenadores no es barato ya que tiene un precio aproximado de 0.01 € por CPU/h, que puede no parecer mucho, pero teniendo en cuenta que suele ser necesario un gran número de procesadores y muchas horas de cálculo en los casos de simulaciones grandes el coste asciende a cantidades considerables y que se financian normalmente con partidas públicas en los casos de personal investigador. Si las optimizaciones pueden conseguir un ahorro en tiempo, esto permitiría poder ejecu-

tar simulaciones más grandes utilizando los mismos recursos o la simulación original más rápido y, por tanto, más barato.

6.1. Metodología

Un aspecto importante que afecta a ambas optimizaciones es que durante el desarrollo del código, para cada cambio realizado se tenía que verificar que este no afectaba al resultado final, no en tiempo o memoria sino en valor numérico con significado físico. Para ello (y también de forma normal), cada ejecución del código genera un archivo de datos entre los que se encuentran magnitudes tanto físicas, como el Reynolds de fricción Re_τ, como computacionales, como el tiempo entre comunicaciones. Para asegurarse de que los cambios eran correctos se ejecuta la versión original, sin ningún cambio hecho, y cuyos resultados se guardan y se catalogan como *verdad*, que serán precisamente los resultados con los que se compararán los producidos por versiones con cambios a fin de que los resultados físicos sean los mismos, pues es la misma simulación.

Esta verificación se realiza mediante un *script* en MATLAB proporcionado por el tutor, en el que se representan las magnitudes anteriormente mencionadas de las versiones deseadas a fin de comparar los resultados. Un ejemplo de esta comparación de una de las simulaciones realizadas se puede ver en la Figura 6.1 donde, en este caso, se grafica el valor de Re_τ a lo largo del tiempo[1], y cuyo valor no es ahora mismo de interés.

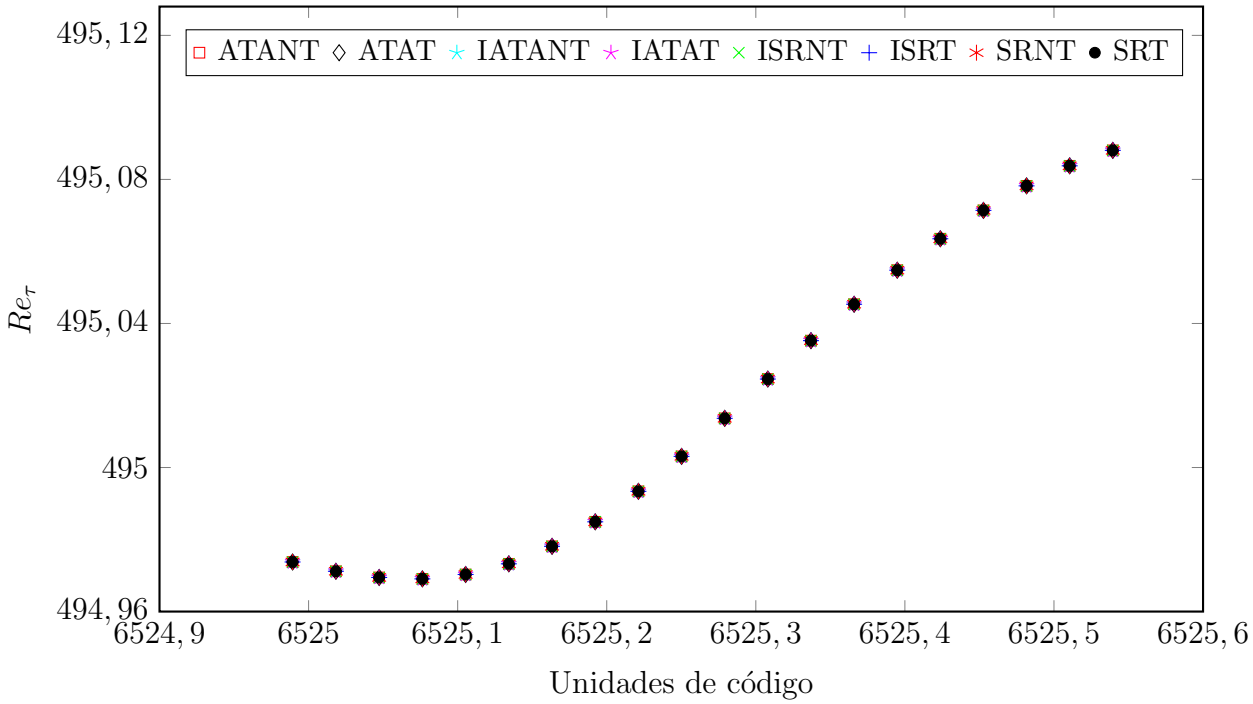


Figura 6.1: Valor de Re_τ a lo largo del tiempo para diferentes versiones del código.

[1]En el eje x se representan las unidades de tiempo en unidades de código, siendo estas una medida del avance de los pasos temporales del código y sin mayor importancia

En este caso el gráfico pertenece a la ejecución del código con las diferentes versiones de comunicaciones, que se pueden observar en la leyenda y que más adelante se explicarán con detalle. Se observa, aunque con dificultad, que todas las versiones producen el mismo resultado, de ahí el solape de los puntos, lo que indicaría que las modificaciones realizadas son correctas.

Los gráficos de las magnitudes físicas únicamente interesan para comprobar que el resultado proporcionado es el correcto. Sin embargo, los que son más interesantes para los objetivos del trabajo son los tiempos de ejecución. De entre estos destacan particularmente tres. En primer lugar encontramos la métrica más importante y que engloba a las dos restantes, el tiempo por paso t_{paso}, que se puede observar en la Figura 6.2[2] y que, como su nombre indica, mide el tiempo total por paso temporal[3].

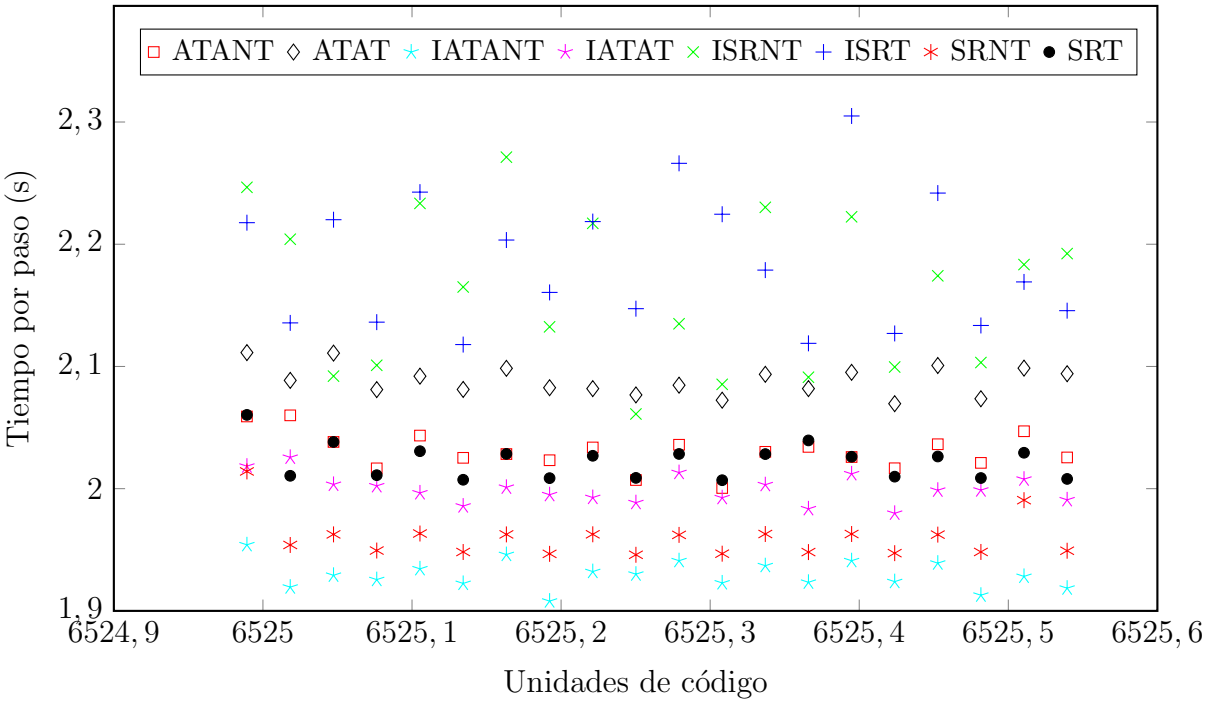


Figura 6.2: Tiempo por paso temporal para las diferentes versiones.

En ella se puede observar como las diferentes rutinas de comunicaciones tienen un tiempo de ejecución diferente, lo que supone que unas son más rápidas que otras. El análisis exhaustivo de los resultados se realizará más adelante.

En segundo lugar de importancia encontramos el tiempo de las comunicaciones t_{com}, que expresa el tiempo que tardan los procesos en enviar y recibir datos, incluyendo las esperas de estos entre medio de la operación. Un ejemplo se puede visualizar en la Figura 6.3.

La última métrica de interés es el tiempo de transpuesta de matrices t_{trans} que, aunque parezca una operación trivial, en campos fluidos grandes puede suponer un porcentaje importante del tiempo total por paso. Un ejemplo se puede visualizar en la Figura 6.4.

[2]Recordemos que son únicamente ejemplos cuyos valores no son importantes en este momento.
[3]El código ejecuta 100 pasos temporales, pues con este número es suficiente, aunque es un parámetro que, como es obvio, se puede variar a elección del usuario.

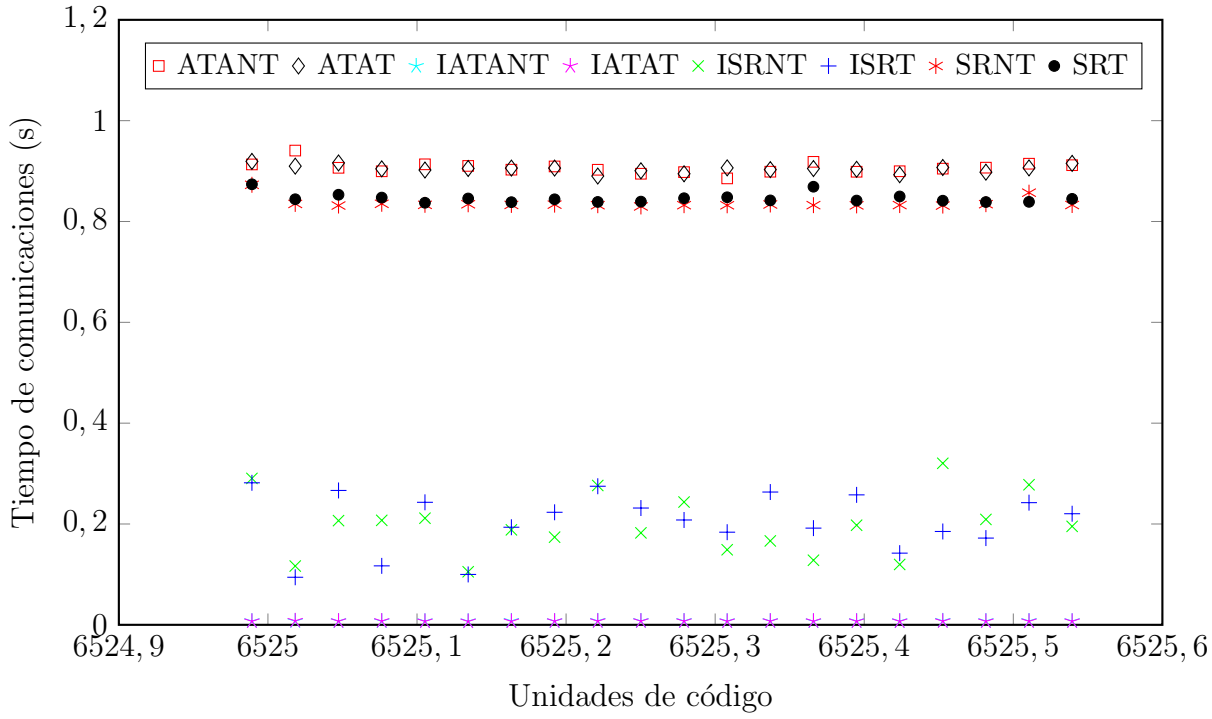


Figura 6.3: Tiempo de comunicaciones por paso para las diferentes versiones.

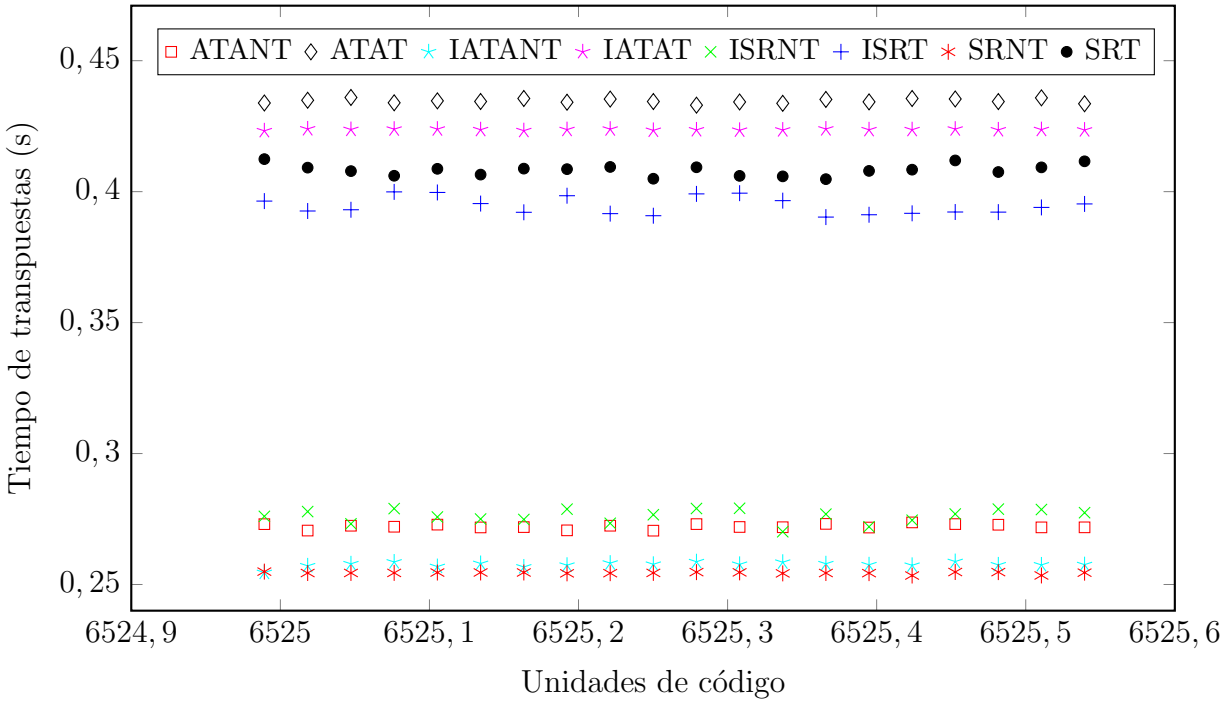


Figura 6.4: Tiempo de transpuestas por paso para las diferentes versiones.

Pese a que estos gráficos presentados son los originales que muestran los diferentes tiempos mencionados, la comparación posterior se realizará mediante diagramas de barras con los tiempos promediados entre todos los pasos y normalizados con la versión más rápida, de manera que sea más sencillo comparar el rendimiento de las diferentes versiones disponibles.

6.2. Cambio a precisión doble

La primera optimización llevada a cabo, que ayudó a consolidar el conocimiento del código, ha sido la de cambiarlo a precisión doble. La versión original ya escribía los datos a disco en precisión doble pero los cálculos se hacían en simple. En FORTRAN la declaración de variables es estática, por lo que se hace necesario definir expresamente en las primeras líneas qué datos contendrá cada variable. En las declaraciones dinámicas, presentes en lenguajes más modernos, una variable dada puede albergar diferentes tipos de datos a lo largo de la ejecución del programa. Para definir de forma correcta las variables se ha de tener en cuenta el tipo de dato que albergarán (cadenas de texto, números reales, complejos, enteros, decimales, etc.) y su tamaño en bytes, siendo por defecto la precisión simple (4 bytes) y pudiendo ampliarlo a doble (8 bytes), cuádruple (16 bytes), etc.

Así pues, al cambiar las rutinas de cálculo del código de simple precisión a doble la memoria RAM necesaria para albergar el dominio del problema con todas las variables ascenderá también al doble de la original, haciendo que muchos de los casos simulables en precisión simple no lo sean en doble por falta de memoria. La cuestión a responder es cuánto más tarda el código en ejecutarse, si el doble, menos o más.

La mayor parte de los cambios, a parte de redefinir las variables al tamaño correcto, ha sido la de cambiar las rutinas provistas por las librerías utilizadas como FFTW3, HDF5 o MPI, que se comentaban en el Capítulo 4. En FFTW3 se han tenido que cambiar las funciones de las FFTs a otras que soportaran precisión doble, para HDF5 se ha cambiado el tipo de dato en su declaración y al leer del campo fluido proporcionado y en MPI se ha cambiado el tipo de dato en las rutinas de comunicaciones así como los tamaños en bytes que sirven como argumentos de entrada necesarios para estas. Esta tarea ha sido sencilla gracias a la excelente documentación de estas librerías, que indicaban de forma clara los cambios necesarios en las funciones a utilizar.

La importancia de este cambio a doble precisión no es otra que la de intentar resolver otros problemas diferentes con tal de avanzar en el estudio de la turbulencia. Uno de los casos ya pensados para un futuro problema es el de poder introducir pequeñas perturbaciones (pequeños incrementos en las magnitudes del fluido) en un flujo estacionario laminar y observar su evolución. Esto en precisión simple no se puede hacer ya que, dado el carácter caótico de la turbulencia y su dependencia sensible a las condiciones iniciales, si se introdujera una perturbación surgirían dos escenarios. El primero es que si esta perturbación es muy pequeña el valor quedaría truncado por el ordenador debido al número de decimales posibles a representar con precisión simple, por lo que esta perturbación no afectaría. El segundo escenario es que, si esta perturbación es lo suficientemente grande como para no ser truncada por el ordenador, esta sea de hecho demasiado grande como para estudiar en detalle la evolución del fluido ya que una variación relativamente grande en las magnitudes del fluido supondrá un cambio muy brusco en el movimiento del mismo, con posiblemente poca información importante a analizar. La importancia de la precisión doble es precisamente la de poder introducir una perturbación del tamaño correcto para el estudio detallado de la evolución del movimiento del fluido.

En la Tabla 6.1 se puede observar el aumento de tiempo que conlleva la ejecución en doble precisión. En ella se puede observar como el tiempo por paso es aproximadamente un 68.5 % de media mayor en el caso de doble precisión, las comunicaciones un 75 % y las transpuestas un 106 %, con las versiones de comunicaciones originales. Como se comentaba antes, el gasto de memoria es el doble, sin embargo, se ve como el tiempo de ejecución es menos del doble. Esta ralentización se puede explicar principalmente por el efecto de las comunicaciones y otras esperas. Uno de los factores importantes es la red de comunicaciones, que interesa que tenga el máximo ancho de banda posible para enviar estos datos entre procesos de forma rápida, ya que el tamaño de datos a enviar es el doble. Esto hace que si la latencia de la red es alta comparada con el ancho de banda este tiempo en comunicaciones se reduzca en porcentaje ya que la comunicación va a durar más tiempo absoluto y la latencia será igual a la original. También puede afectar el tiempo de espera entre procesadores para efectuar las comunicaciones, que puede ser igual al original, pero dado que las comunicaciones tardan más esta espera es menor en porcentaje, lo que también se refleja. Si observamos el tiempo de transpuestas, vemos como es aproximadamente el doble al original. Aquí no hay nada que hacer ya que el tamaño de los datos que se mueven es el doble por lo que el coste temporal también es el doble.

Versión	SendRecv	Alltoall
Tiempo paso	68 %	67 %
Tiempo comunicaciones	77 %	73 %
Tiempo transpuestas	113 %	99 %

Tabla 6.1: Aumento de tiempos al cambiar de simple a doble precisión.

6.3. Optimización comunicaciones

El intento de optimizar los algoritmos relacionados con las comunicaciones era la parte de mayor interés en la realización de este proyecto. Se han desarrollado seis rutinas nuevas, que se añaden a las dos originales, con el fin de compararlas y determinar cuál es la más rápida dependiendo del tamaño del dominio.

Como se comentaba en el capítulo anterior, los aspectos a destacar involucrados en las comunicaciones eran, por una parte, si la comunicación es punto a punto o colectiva y, por otra parte, si esta es *blocking* o no. Además, para la estructura del algoritmo a ejecutar expuesta en la Sección 5.1 también destaca el hecho de las matrices transpuestas locales. Todos estos aspectos se han tenido muy en cuenta a la hora de realizar las nuevas rutinas, haciendo una especie de estudio paramétrico entre las varias opciones disponibles, que se detallan a continuación.

- **SRT**: Esta rutina pertenece al código original y utiliza como función de comunicaciones *MPI_SENDRECV* en su forma *blocking*, explicada en el capítulo anterior. Además, en esta rutina también se realizan las matrices transpuestas locales para ordenar los datos y que estén contiguos en memoria.

- **ISRT**: Esta rutina es una de las nuevas desarrolladas y utiliza como función de comunicaciones *MPI_ISENDRECV* en su forma *non-blocking*, de manera que se intercala trabajo entre las comunicaciones a fin de aprovechar el tiempo muerto en el que los procesadores no se están comunicando. Esta rutina también realiza las matrices transpuestas locales para ordenar los datos.

- **SRNT**: Esta rutina es una de las nuevas desarrolladas y utiliza como función de comunicaciones *MPI_SENDRECV* en su forma *blocking*, como la original. La diferencia es que esta versión no realiza las matrices transpuestas para ordenar los datos sino que únicamente se añaden los elementos intermedios necesarios (ceros numéricos) y la siguiente operación lee estos de forma no contigua en memoria.

- **ISRNT**: Esta rutina es una de las nuevas desarrolladas y combina los dos cambios de las dos rutinas previas, siendo la función de comunicaciones *MPI_ISENDRECV* en su forma *non-blocking* y sin ejecutar las transpuestas locales sino leyendo desordenado en la operación siguiente.

- **ATAT**: Esta rutina pertenece al código original y utiliza como función de comunicaciones *MPI_ALLTOALLv* en su forma *blocking*, explicada también en el capítulo anterior. Además, en esta rutina se realizan las matrices transpuestas locales para ordenar los datos y que estén contiguos en memoria.

- **IATAT**: Esta rutina es una de las nuevas desarrolladas y utiliza como función de comunicaciones *MPI_IALLTOALLv* en su forma *non-blocking*, de manera que se intercala trabajo entre las comunicaciones. Esta rutina también realiza las matrices transpuestas locales para ordenar los datos.

- **ATANT**: Esta rutina es una de las nuevas desarrolladas y utiliza como función de comunicaciones *MPI_ALLTOALLv* en su forma *blocking*. Esta versión no realiza las matrices transpuestas para ordenar los datos.

- **IATANT**: Esta rutina es una de las nuevas desarrolladas y combina los dos cambios de las dos rutinas previas, siendo la función de comunicaciones *MPI_IALLTOALLv* en su forma *non-blocking* y sin ejecutar las transpuestas locales.

Para evaluar el rendimiento de todas ellas se han ejecutado para diferentes tamaños de campos fluidos y procesadores, cuyas características pueden observarse en la Tabla 6.2. Los tamaños N_x, N_y, N_z que se muestran están en el espacio físico, no el de Fourier, siendo N el número total de puntos. Sin embargo, por la arquitectura del código, el número máximo de procesadores a utilizar corresponde con el número de planos complejos en x en el dominio de Fourier M_x, de manera que $M_x = \frac{1}{2}\frac{2}{3}N_x$, donde el $1/2$ corresponde a reducir el número de elementos en dos ya que se consideran planos complejos, con el doble de datos por elemento que uno simple, y el $2/3$ corresponde a la inversa del factor de expansión para evitar el *aliasing* al pasar del dominio de Fourier al dominio físico, ya que en este caso queremos calcular el tamaño del dominio de Fourier, que es el inicial. Las diferentes simulaciones que así lo permitan por su tamaño no solo han sido ejecutadas con el número máximo de procesadores M_x sino también con un número menor para poder analizar el escalado, lo que se muestra en la última columna de la tabla antes mencionada.

Re_τ	N_x	N_y	N_z	N	M_x	Procesadores
180	384	251	288	$3 \cdot 10^7$	128	128
500	1536	251	1152	$4 \cdot 10^8$	512	128, 256 y 512
1000	3072	383	2304	$3 \cdot 10^9$	1024	512 y 1024
10000	6144	2101	6144	$8 \cdot 10^{10}$	2048	2048

Tabla 6.2: Tamaños de las mallas utilizadas.

Por último, notar que la utilización de los diferentes campos fluidos de diferente tamaño es una práctica necesaria debido a que, para campos pequeños la arquitectura del código no permite utilizar todos los procesadores deseados sino únicamente los correspondientes al número de planos y, para campos grandes la utilización de un menor número de procesadores haría que la simulación tardara mucho tiempo y resultara irrealizable debido a los requerimientos de memoria RAM. La simulación de cada campo necesita un número diferente de pasos temporales para completarse y obtener toda la información relacionada con la turbulencia a estudiar. En este trabajo solo se realizan 100 pasos temporales por simulación, pues son suficientes para los análisis que se pretenden realizar.

Para tener un orden de magnitud de los requisitos en memoria a los que a lo largo del trabajo se ha ido haciendo referencia, si consideramos el campo más grande de los anteriores, de $8 \cdot 10^{10}$ puntos de malla, en doble precisión (8 bytes por elemento) y con una media de 10 variables usadas (las tres velocidades, las tres vorticidades, ϕ y variables auxiliares) esto da un total de 5.8 TB de memoria RAM, tal y como se puede ver en (6.1). Este valor debe repartirse entre todos los nodos utilizados y en superordenadores punteros hay nodos especiales donde la memoria disponible es elevada por lo que no supone, *a priori*, un problema.

$$\text{Memoria} = 8 \cdot 10^{10} \cdot 8 \cdot 10 = 6{,}4 \cdot 10^{12}\ bytes = 5{,}8\ TB \tag{6.1}$$

6.4. Resultados

Un análisis exhaustivo de los datos obtenidos no es en absoluto trivial debido al gran número de variables involucradas en las diferentes versiones a ejecutar. Las simulaciones anteriormente comentadas han sido lanzadas en MareNostrum (MN) pero principalmente en SuperMUC (SM), por lo que también será posible comparar ambas máquinas. Debido a la naturaleza de las simulaciones DNS, nos interesa optimizar el coste temporal principalmente en las simulaciones más grandes, aunque también en las intermedias para estudiar diferentes tipos de problemas. Todas las simulaciones han sido realizadas en la versión de doble precisión, más costosa que en simple tal y como se veía en secciones anteriores. Para poder comparar de forma más intuitiva las diferentes versiones no se utilizarán los valores de tiempo absoluto (aunque se mostrarán más adelante para tener un orden de magnitud) sino que se normalizarán todos con los de la versión más rápida, facilitando así su evaluación.

Para comenzar con el análisis podemos observar la Figura 6.5, que muestra un ejemplo de una simulación para $Re_\tau = 1000$ con 1024 procesadores. En ella se puede observar, mediante las barras del tiempo por paso, como la versión más rápida corresponde al caso

SRNT, que se muestra en el gráfico con un valor del 100 % y sobre la que se normalizarán las demás. La segunda versión más rápida sería para este caso SRT, con valor de 104 %, lo que indica que es un 4 % más lento que la versión más rápida. En esta figura también se pueden observar otras cosas importantes que se repiten prácticamente en todos los casos simulados.

La primera es que las comunicaciones en los casos *non-blocking* no se pueden medir correctamente ya que se sabe el tiempo que tarda en lanzarse, que es el que se mide, pero no el tiempo total que tarda en finalizarse la comunicación (lanzar y recibir), por lo que el análisis no es directo y es necesario centrarse únicamente en estos casos en el tiempo por paso. Se observa que estas versiones no muestran resultados buenos, quedando prácticamente en su totalidad como las versiones más lentas. Hablando de las propias funciones de comunicaciones, se puede observar como las versiones con el algoritmo del hipercubo de *SENDRECV* son más rápidas que las de *ALLTOALL* (56 % vs 67 %), ambas en su versión *blocking*. Este hecho se da en todos los casos simulados excepto en el más pequeño y puede ser debido al cuello de botella que la comunicación global puede producir, ya que todos los procesos ejecutan esta orden prácticamente a la vez por lo que la red de conexión puede saturarse. También es importante destacar que las comunicaciones suponen, para los casos más rápidos, un 56 % del tiempo total del tiempo por paso, una cantidad muy alta y que para los menos familiarizados en estos ámbitos puede sorprender, ya que se tarda más en comunicar datos entre procesadores que en hacer los propios cálculos de resolución.

El segundo aspecto importante a observar en esta figura es el caso de las versiones con y sin transpuestas. Se puede observar como las versiones sin transpuesta, como es obvio, tardan menos tiempo en realizar las indispensables (13 % vs 20 %). Este tiempo no es nulo ya que se hace necesario en ese punto del algoritmo añadir conjuntos de ceros en posiciones concretas, previos a la realización de la FFT de Fourier a físico para evitar el *aliasing*. Sin embargo, *a priori*, aunque se sepa que no realizar la transpuesta puede que ahorre tiempo, el hecho de leer los datos de forma no contigua en memoria puede que sea más costoso, por lo que se tenía que analizar. Este aspecto es el determinante de que, por ejemplo en este caso, gane la versión sin transpuestas (SRNT) a la de con transpuestas (SRT), quedando ambas a la cabeza de las versiones. Se puede observar como las comunicaciones son idénticas por lo que el factor diferencial ha sido el de las transpuestas, con un 7 % de diferencia entre ellas que se convierten en un 4 % en el tiempo total por paso, siendo ese 3 % de diferencia el comentado, el hecho de leer los datos de manera no contigua en memoria.

Los resultados más importantes, en líneas generales, se pueden observar en la Tabla 6.3, que se centra principalmente en la versión más rápida para cada campo fluido simulado para ambos superordenadores. Así, será posible analizar la escalabilidad del código al comparar los campos de diferente tamaño y la versión más rápida en cada uno, así como el propio rendimiento de las máquinas, siendo esto último más por curiosidad. Los campos con guiones (-) son debido a que no todas las simulaciones han sido ejecutadas en ambas máquinas, por lo que no se disponen de los datos, aunque son únicamente dos casos y no afecta a la comparación general.

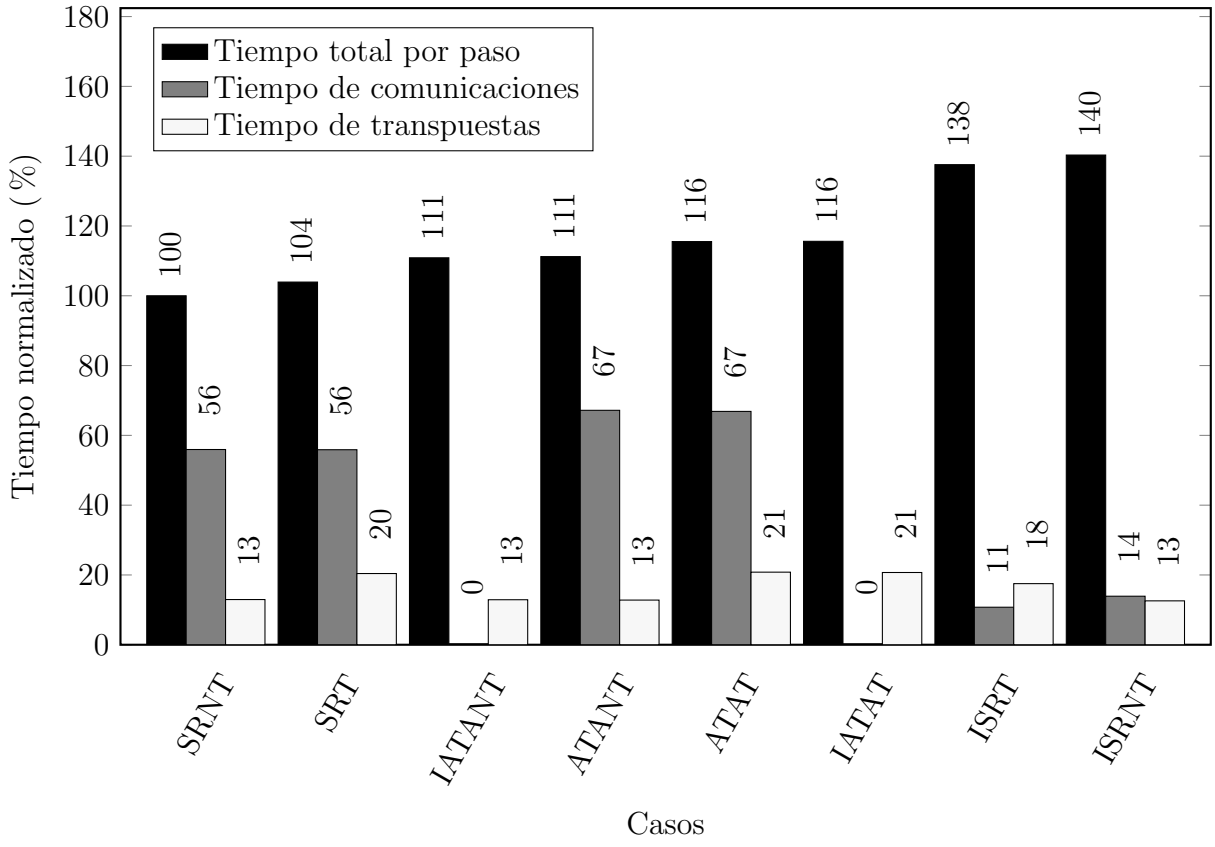


Figura 6.5: Comparación de tiempos normalizados para $Re_\tau = 1000$ con 1024 procesadores.

Re_τ_Procesadores Numeración del caso		180_128 1	500_128 2	500_256 3	500_512 4	1000_512 5	1000_1024 6	10000_2048 7
Caso más rápido	MN	ATAT	-	IATANT	SRNT	SRNT	SRNT	-
	SM	SRT	ATANT	SRNT	SRNT	SRNT	SRNT	SRT
Tiempo paso (s)	MN	0.08	-	1.93	1.24	6.54	4.42	-
	SM	0.08	3.35	1.99	1.16	6.30	3.59	38.31
Tiempo paso por millón de elementos (s/M)	MN	0.0027	-	0.0048	0.0031	0.0022	0.0015	-
	SM	0.0027	0.0084	0.0050	0.0029	0.0021	0.0012	0.0005
Tiempo comunicaciones / Tiempo paso	MN	58 %	-	0.4 %	55 %	46 %	58 %	-
	SM	62 %	41 %	50 %	57 %	51 %	57 %	50 %
Tiempo transpuestas / Tiempo paso	SM	8 %	17 %	14 %	12 %	13 %	12 %	16 %

Tabla 6.3: Comparación entre los campos de diferente tamaño.

En la tabla anterior se puede observar, en la segunda fila, como para el caso más pequeño y el más grande simulado las versiones ganadoras pertenecen a las originales mientras que para los demás casos intermedios la versión más rápida es, prácticamente en sus totalidad, la de SRNT. Al final del capítulo se mostrará de manera cuantitativa en qué casos se ha conseguido optimizar el código y cuánto respecto a la versión original. En la siguiente fila se muestra el tiempo absoluto del tiempo por paso, en segundos, para que el lector tenga un orden de magnitud del coste temporal de ejecución de estos códigos[4]. Vemos como estos tiempos varían desde las centésimas de segundo a los propios segundos, siendo el máximo casi 40 segundos por paso para el caso más grande. Esta información también nos permite comparar las dos máquinas donde se han ejecutado los casos y afirmar que SuperMUC es algo más rápido que MareNostrum, entre un 5 % y 20 % dependiendo del caso, principalmente en los grandes. Esto es debido tanto a los procesadores, que efectivamente son más potentes los de SuperMUC tal y como se muestra en sus manuales técnicos [28] [29], como por la red de conexión, que puede ser diferente tal y como se observa en los casos 1 y 3, donde para el mismo caso las versiones más rápidas son diferentes dependiendo de la máquina.

Para poder evaluar la escalabilidad, a parte de las métricas de aceleración y eficiencia que se verán más adelante, dado que el tamaño de los campos así como el número de procesadores varían según el caso no es posible compararlos de manera directa por lo que se opta por comparar el tiempo por paso por millón de elementos. Esta métrica ha sido elegida debido a que el tamaño de la simulación y por tanto su coste computacional está directamente relacionado con el número de puntos de discretización de la malla del dominio. Así, se puede observar como la evolución es relativamente buena ya que a mayor número de elementos según el caso con sus procesadores máximos permitidos por la propia arquitectura del código el tiempo por paso es menor, lo que significa que, aunque en los casos más grandes el tiempo absoluto incremente respecto a los más pequeños en órdenes de magnitud, y aunque el tamaño de la simulación también lo haga, no incrementa de forma prohibitiva, lo que puede permitir simulaciones más grandes en el futuro siempre que se consiga el número de procesadores y el generoso tiempo de computación necesario. Recordemos que el último caso, de $Re_\tau = 10000$, es el más grande simulado hasta la fecha.

Para acabar con la Tabla 6.3, las dos últimas filas muestran el tiempo de comunicaciones y transpuestas en porcentaje respecto del tiempo por paso. Como se expresaba anteriormente, el porcentaje de las comunicaciones es muy elevado, rondando el 50 % del tiempo total. Respecto a esto, se puede observar como en los campos en los que es posible ejecutar la simulación con un número de procesadores y el doble de este valor el porcentaje del caso con menos procesadores es menor que el caso con el doble de procesadores. Esto es porque, en comparación, el resto de operaciones a realizar se ejecutan de manera más lenta con menos procesadores. Sin embargo, si se toma el tiempo absoluto, este es obviamente mayor con menos procesadores ya que cada uno tiene que enviar más datos y realizar más operaciones. En cuanto a las transpuestas, se muestran únicamente los datos de SuperMUC ya que las ejecuciones en MareNostrum contenían un pequeño fallo al medir este dato, aunque las conclusiones son las mismas. Se observa como el porcentaje del coste temporal ronda el 10-15 %, dependiendo de si se trata de las versiones con o sin

[4]Todos los resultados están promediados entre los 100 pasos temporales ejecutados. Una simulación completa, dependiendo del tamaño del campo, ronda un número de pasos temporales del orden de 10^4 a 10^6, aunque puede variar.

transpuestas. Como veíamos en el capítulo anterior, dos de los aspectos fundamentales en esta clase de códigos son las comunicaciones y las transpuestas, que juntos pueden llegar a consumir hasta un 65 % - 70 % del tiempo total de ejecución, cifra muy sorprendente.

Una comparación de todas las versiones y su tiempo por paso respecto a la versión más rápida para los diferentes campos puede observarse en la Tabla 6.5 para los casos ejecutados en MareNostrum, donde la versión ganadora se muestra con un 0. En ella se puede observar como la versión SRNT gana en 3 de los 5 casos. Las diferentes versiones difieren en todos los casos entorno a un máximo del 10 %. Sin embargo, para los casos de ISRT y ISRNT estas diferencias son muy notables, llegando a ser hasta un 100 % más lento para el caso 5. Esto puede ser debido a que la red de comunicaciones se satura al mandar todos los datos seguidos antes de recibir ninguno ya que ambas son de la forma *non-blocking* de la función *SENDRECV*. Al comparar los casos 5 y 6 se puede inferir que esto es debido al uso de un número menor de procesadores para el mismo campo, ya que en el caso 6 la diferencia con la versión más rápida ronda el 25 %, cifra alta pero menor que la anterior. Respecto a los resultados en SuperMUC, mostrados en la Tabla 6.6, los resultados son bastante similares, con unas diferencias entre versiones de entorno a un máximo del 25 %, exceptuando de nuevo los casos de ISRT y ISRNT, que en los casos donde hay muchos procesadores (a partir de 512) el algoritmo se ve sustancialmente ralentizado por la gran cantidad de comunicaciones seguidas que saturan la red de conexión. De nuevo SRNT gana en 4 de los 7 casos, y las originales en el campo más pequeño y el más grande.

Siguiendo con la comparación de la escalabilidad, en la Tabla 6.4 se pueden observar las métricas clásicas de aceleración o *speedup* y eficiencia para los casos en los que ha sido posible utilizar el mismo campo fluido utilizado con distinto número de procesadores. Para la aceleración se toma como valor inicial el obtenido con el menor número de procesadores y no el valor unidad con el que se define de manera clásica ya que directamente sería imposible simularlo. Si recordamos, la máxima aceleración esperable era el factor de aumento del número procesadores. Así, si lanzamos la ejecución con 128 y 256 procesadores la aceleración máxima será de 2, por ser el número de procesadores el doble. Se puede observar en la fila correspondiente de la tabla que estos valores de aceleración son bastante buenos, proporcionando mejoras desde un 48 % a un 91 % al ejecutar las simulaciones con el doble de procesadores que en el caso original, siendo el incremento máximo posible del 100 %. En cuanto a la eficiencia, que medía la desviación respecto a la aceleración ideal, vemos que también es relativamente buena, con valores superiores al 75 % para la ejecución con el doble de procesadores que los iniciales y en torno al 50 % para la ejecución con 4 veces los originales. Ambas métricas permiten confirmar que el código escala de manera decente aunque menos eficiente cuanto más grande es el campo.

Casos	$Re_\tau = 500$					$Re_\tau = 1000$			
Ordenador	MN		SM			MN		SM	
Procesadores	256	512	128	256	512	512	1024	512	1024
Speedup	1	1.56	1	1.91	2.18	1	1.48	1	1.75
Eficiencia (%)	100 %	78 %	100 %	96 %	55 %	100 %	74 %	100 %	88 %

Tabla 6.4: Métricas de rendimiento de los casos posibles.

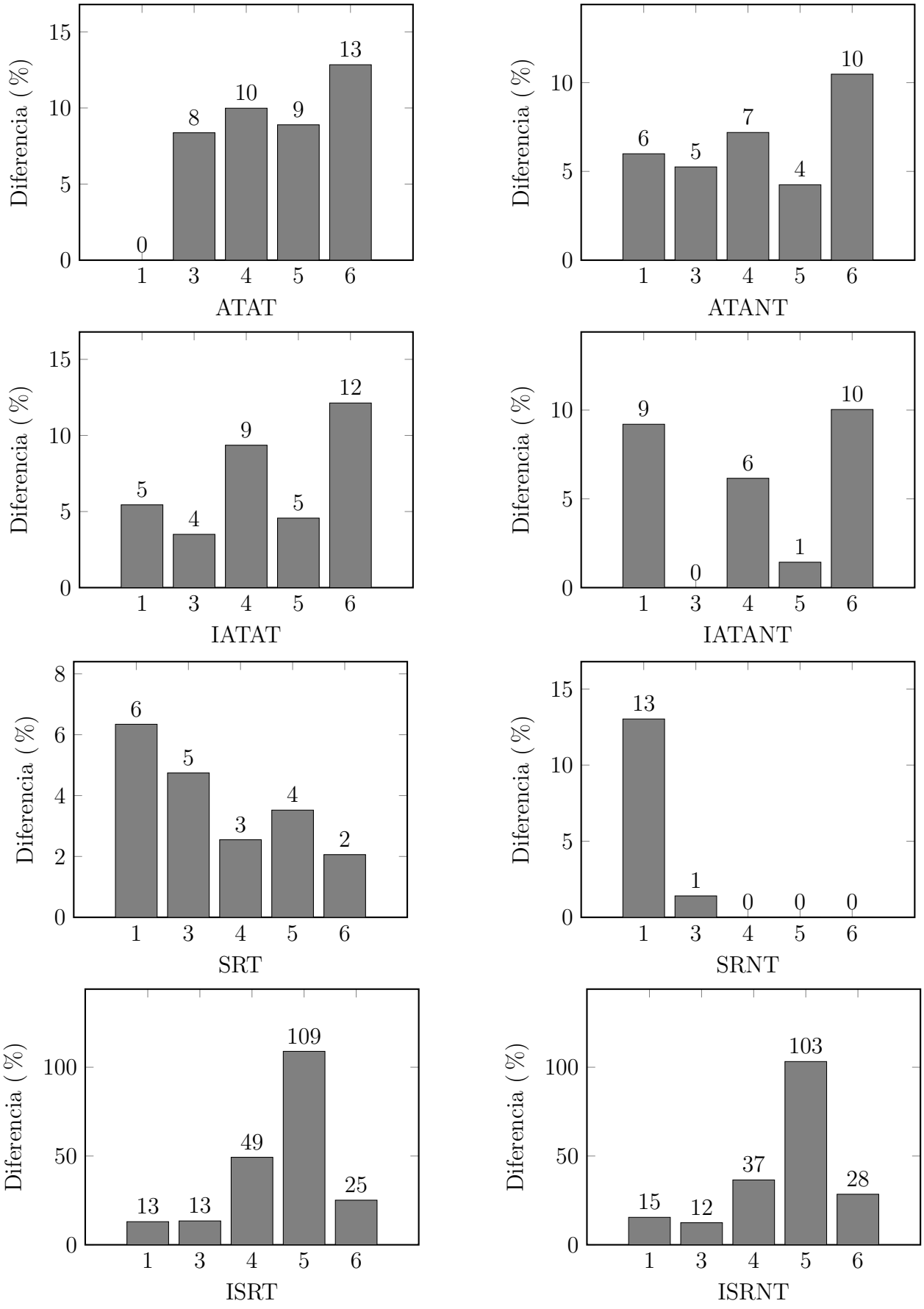


Tabla 6.5: Diferencia del tiempo por paso entre cada versión y la versión más rápida para los diferentes casos en MareNostrum.

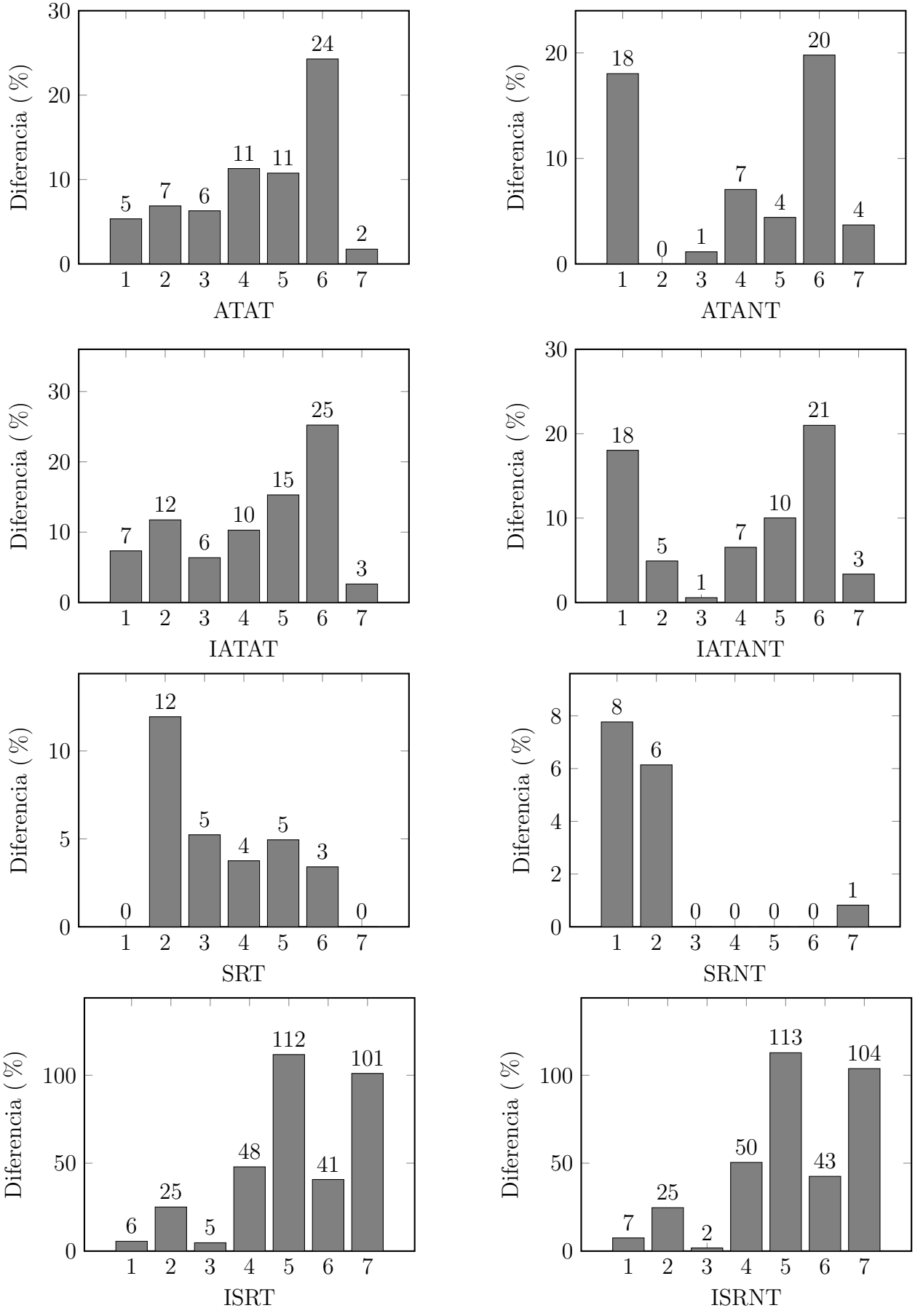


Tabla 6.6: Diferencia del tiempo por paso entre cada versión y la versión más rápida para los diferentes casos en SuperMUC.

Debido a que en SuperMUC se ha podido disponer de más tiempo y recursos de cómputo otros análisis a destacar han sido realizados en esta máquina, los cuales se muestran a continuación. En primer lugar, se han podido ejecutar algunos casos con 24 y 48 procesos MPI por nodo (donde cada nodo tiene 48 núcleos y 48 hilos sin usar el *hyper-threading*) para ver las diferencias, que se encuentran plasmadas en la Tabla 6.7 (todas las tablas anteriores reflejan los resultados de la ejecución con 48). Es importante destacar que el número total de procesos con los que se ejecuta la simulación es el mismo, lo único que cambia es el hecho de utilizar más nodos y un número diferente de núcleos en cada caso, siendo un proceso por núcleo en el caso de 48 procesos y 2 núcleos por proceso en el caso de 24 procesos. Este análisis no ha podido realizarse para todos los casos ya que valores elevados de procesadores necesarios ocuparían una cantidad de nodos muy grande en la máquina por lo que el coste monetario sería también superior y su accesibilidad difícil. En la tabla mencionada se puede observar como ejecutar los campos con 24 procesos por nodo es aproximadamente un 30 % más rápido en el tiempo por paso que ejecutarlos con 48, manteniendo el total similar. Este beneficio es debido a dos aspectos. El primero es que para el caso de 24 procesos la red de comunicación está menos saturada que en el caso de 48 ya que se envían la mitad de datos desde el mismo nodo. Esto hace que las comunicaciones sean aproximadamente un 40 % más rápidas ya que no se interfieren entre ellas. El segundo aspecto es el hecho de que para el caso con 24 procesos se disponen de 24 núcleos adicionales (48 en total) para realizar las operaciones que puedan ser paralelizadas con OpenMP, como es el caso de las transpuestas o FFTs, por lo que se cuenta con recursos computacionales adicionales. Así, se puede observar como las transpuestas son aproximadamente de un 40 % a un 60 % más rápidas. Se hace necesario recalcar que el coste monetario que supondría esta ejecución para valores más elevados de procesadores es muy alto, utilizando el doble de recursos computacionales, por lo que se tendría que evaluar antes de realizar la ejecución final aunque se observa que el doble de recursos computacionales no hace la ejecución el doble de rápida sino un 30 % aproximadamente debido a la naturaleza del código.

Re_τ_Procesadores	180_128	500_128	500_256
Tiempo paso	33 %	25 %	34 %
Tiempo comunicaciones	42 %	31 %	47 %
Tiempo transpuestas	48 %	60 %	38 %

Tabla 6.7: Mejora de ejecutar con grupos de 24 en vez de 48 procesos por nodo.

Otro análisis relacionado con la ejecución en grupos de 24 y 48 procesos recién comentada es el factor del caché *blocking*. Como se veía en el capítulo anterior es posible implementar las transpuestas con caché *blocking* simple o doble, que es precisamente lo que se compara en la Tabla 6.8. En ella se observa como la mejora de usar *blocking* doble respecto al simple es notoria en algunos casos, llegando hasta un 4 % de mejora para el caso más grande, una mejora bastante buena para la simplicidad que implica implementarla.

Re_τ_Procesadores	180_128	500_128	500_256	500_512	1000_512	1000_1024	10000_2048
Grupos de 24	1.7 %	3.8 %	1.9 %	2.4 %	-	-	-
Grupos de 48	0.3 %	4.3 %	0.7 %	1.6 %	3.3 %	2.2 %	4.2 %

Tabla 6.8: Mejora de usar caché blocking doble respecto a simple.

Otra comparación que se puede hacer y enfocada más bien a la escalabilidad en líneas generales no esta vez del código si no de las características de una DNS como las vistas anteriormente en la Tabla 3.1 se puede encontrar en la Tabla 6.9. En ella se puede ver, por ejemplo, como para conseguir un Número de Reynolds 50 veces superior es necesario una malla 2500 veces más grande, incrementándose el tiempo por paso en un factor de 500. Recordemos que a mayor Reynolds mayor número de pasos temporales son necesarios.

Casos	$Re_\tau = 180$	$Re_\tau = 500$	$Re_\tau = 1000$	$Re_\tau = 10000$
Diferencia puntos	-	13	100	2667
Diferencia Reynolds	-	2.7	5.6	55.6
Diferencia tiempo	-	14.5	44.9	478.9

Tabla 6.9: Comparación de características respecto del caso más pequeño para la ejecución ideal de un plano por procesador en la versión más rápida en cada caso.

Finalmente se está en condiciones de evaluar si de verdad se ha conseguido o no optimizar el código original mediante las técnicas que se han desarrollado a lo largo del trabajo. En la Tabla 6.10 se puede ver este porcentaje de optimización para cada uno de los casos simulados. Cabe destacar que en estas mejoras no se incluyen las ejecuciones con un menor número de procesos por nodo sino que emergen del hecho de combinar la posible mejora de la versión de comunicaciones respecto a la original más rápida, que recordemos era SRT, con la posible mejora del uso del caché *blocking* doble, que no se utilizaba en la versión original en la que se utilizaba el *blocking* simple. Así, se puede observar como, exceptuando el primer caso más pequeño, se ha conseguido optimizar el código en cantidades que van desde un 4 % a un 10 %. En todos los casos la optimización es debida a ambos aspectos, las comunicaciones y el *blocking*. Sin embargo, en el caso más grande la mejora es únicamente debido al *blocking* y no a las comunicaciones. Estas mejoras pueden llegar a ahorrar recursos monetarios en las ejecuciones finales. Por ejemplo, para el caso del campo más grande, si asumimos que se necesitan aproximadamente 600000 pasos temporales, cifra bastante aproximada a la realidad, con el tiempo por paso visto anteriormente (38 s vs 40 s aproximadamente) la versión mejorada ahorraría aproximadamente de 10 a 15 días de cómputo. Teniendo en cuenta que este campo debe ejecutarse con 2048 procesadores y que el precio aproximado es de 0.01 € por CPU/h resultaría en un ahorro monetario de unos 7000 € en recursos computacionales.

Re_τ_Procesadores	180_128	500_128	500_256	500_512	1000_512	1000_1024	10000_2048
Optimización total	0.3 %	10.1 %	6.8 %	5.2 %	7.9 %	5.9 %	4 %

Tabla 6.10: Optimización conseguida para cada caso respecto a la versión original.

Capítulo 7

Conclusiones

Una vez finalizado todo el desarrollo del trabajo llevado a acabo es momento de evaluar si los objetivos que se presentaban al principio han sido conseguidos. Además, también se contemplan posibles trabajos futuros.

A lo largo de los diferentes capítulos del proyecto se han ido desarrollando en profundidad los diferentes aspectos que intervienen en los códigos de simulación DNS. Por un lado, los aspectos físicos, centrándose en el fenómeno de la turbulencia. Por otro lado, los aspectos numéricos, con la exposición del esquema numérico y discretizaciones que el código de simulación emplea. Por último, los aspectos computacionales, con el desarrollo de los fundamentos que rodean la computación de altas prestaciones y que son tan importantes como los otros dos, pues son los que hacen posible las simulaciones en los superordenadores. Dentro de estos últimos, se han presentado los diferentes problemas principales que afectan concretamente a nuestro código de simulación DNS, destacando entre ellos la división del dominio computacional, la paralelización de las operaciones, las comunicaciones (que consumen un 50 % del tiempo total), las transpuestas (que consumen un 15 % del tiempo total) o la gestión de memoria (llegando a valores muy altos como pueden ser 6 TB de memoria RAM para el campo más grande simulado).

En cuanto a las optimizaciones, en primer lugar, se ha conseguido cambiar el código a precisión doble, con los beneficios que esto puede suponer en cuanto a la simulación de nuevos problemas que ayuden a la comprensión de la turbulencia. Se ha visto como el consumo de memoria RAM es el doble, aunque no así su aumento en el tiempo de ejecución, que es aproximadamente un 70 % superior.

En segundo lugar, se han explorado 6 nuevas alternativas para el algoritmo principal de las comunicaciones. Los resultados indican que efectivamente se ha conseguido optimizar el código en la mayoría de casos, con valores que van desde un 4 % a un 10 %, siendo mayoritariamente la versión más rápida la de SRNT, que combina la función *SENDRECV* con el algoritmo del hipercubo (la cual resulta ser más rápida que la función *ALLTOALL* proporcionada por las librerías) con el caché *blocking* doble para las matrices transpuestas, las cuales se evitan en la medida de lo posible mediante la lectura de operaciones posteriores de manera no contigua en memoria. Otras alternativas que *a priori* se creía que pudieran producir buenos resultados, las comunicaciones *non-blocking*, han resultado ser las peores para los casos más grandes en los que un gran número de procesadores estaba presente (a partir de 512) debido a la saturación de la red de comunicaciones. Estas

optimizaciones permitirán realizar de forma más eficiente las simulaciones en un futuro, teniendo la posibilidad de utilizar un menor número de recursos computacionales al ser estas más rápidas y, por tanto, producir así un ahorro de recursos económicos.

Por último, en cuanto a posibles trabajos futuros, indicar que las optimizaciones de códigos de programación son tareas que nunca finalizan. Siempre hay hueco para nuevas ideas que probar y analizar. Además, el *hardware* de los ordenadores avanza muy rápido por lo que nuevas optimizaciones podrán en un futuro ser realizadas para adaptarse y utilizar de manera óptima los recursos disponibles. Algunas de las ideas para estas optimizaciones es el hecho de rediseñar la arquitectura del código de simulación DNS, cambiando la topología computacional del problema, ya sea dividiendo el dominio de otra forma o pensando en nuevos algoritmos de comunicaciones.

Parte II

Pliego de condiciones y presupuesto

Capítulo 8

Pliego de condiciones

Todo proyecto debe cumplir con unas exigencias técnicas y legales que aseguren su correcta elaboración y que miren por el bienestar de los trabajadores. Estas exigencias incluyen el estado del lugar de trabajo, los recursos y materiales utilizados y las condiciones de seguridad del trabajador, entre otras. Se excluyen de estas exigencias, por tener otras diferentes, proyectos relacionados con instalaciones hídricas, eléctricas o civiles. Este capítulo tiene como objetivo el de presentar de una manera más detallada todas esas exigencias [30].

8.1. Condiciones generales

Para el correcto desarrollo del proyecto es necesario delimitar las tareas y funciones que corresponden a cada una de las partes participantes y su relación.

Funciones a desempeñar por el ingeniero

Entre las tareas a realizar por el ingeniero se encuentran:

- Planificar de manera conjunta con el tutor las diferentes etapas de las que constará el proyecto, incluyendo las tareas a realizar en cada una de esas etapas para asegurar un mínimo de calidad y teniendo en cuenta los objetivos que se busca alcanzar.
- Llevar a cabo un proceso de investigación bibliográfica con el fin de adquirir los fundamentos necesarios que estén presentes en el proyecto. Entre estos recursos se incluyen libros, artículos científicos o material desarrollado por el tutor.
- Redacción de la memoria del proyecto y su presentación ante un tribunal de evaluación de ambas.
- Seguir las órdenes dictadas por el tutor en cuanto al desarrollo específico del proyecto y el análisis de los resultados.

Funciones a desempeñar por el tutor

Entre las tareas a realizar por el tutor del proyecto se encuentran:

- Planificar de manera conjunta con el ingeniero las diferentes etapas de las que constará el proyecto, incluyendo las tareas a realizar en cada una de esas etapas para

asegurar un mínimo de calidad y teniendo en cuenta los objetivos que se buscan alcanzar.

- Introducir al ingeniero técnicas o conocimientos específicos asesorándolo con un mínimo de dedicación ante cualquier duda que pueda surgir durante todo el desarrollo del proyecto.
- Proporcionar los recursos materiales o técnicos para la correcta realización del proyecto.
- Resolver junto al ingeniero los fallos en la ejecución del proyecto de manera que se alcancen los resultados que se buscan alcanzar y verificar el análisis de estos.

8.2. Condiciones particulares

Al desarrollar un proyecto el trabajador debe hacerlo bajo unas condiciones específicas que aseguren que, tanto su salud como su rendimiento son los correctos. Para ello se deben regular las exigencias técnicas mediante legislación, todo para proteger la integridad del trabajador. Estas condiciones se recogen en el Real Decreto 486/1997 y 488/1997 del 14 de abril sobre las disposiciones mínimas de seguridad y salud relacionadas con el espacio de trabajo en el que se incluyen equipos informáticos. En él se presentan directrices con respecto a varias características presentes en los espacios de trabajo.

Orden y limpieza

Las directrices relativas al orden y la limpieza específicas para entornos de trabajo que incluyen equipos informáticos son:

- Mantener el puesto de trabajo limpio y de forma adecuada para desempeñar las tareas. Evitando la acumulación de suciedad o polvo cerca de los equipos. Los suelos deben también permanecer limpios.
- Vía de fácil comunicación hacia un superior de averías o fallos en los equipos informáticos así como de mobiliario común.
- No acumular innecesariamente ni sobrecargar mobiliario como estanterías, zonas de almacenamiento o la superficie de trabajo.
- Correcta gestión de los residuos generados a nivel tanto individual como colectivo.
- Correcto mantenimiento y limpieza de zonas de paso como escaleras además de los equipos de emergencia como extintores o desfibriladores.

Temperatura, humedad y ventilación

Las condiciones ambientales pueden suponer un factor de riesgo o molestia para el trabajador, así, toda la instalación debe estar correctamente aislada térmicamente y contar con una instalación de climatización. Además, se debe evitar:

- Humedad y temperaturas extremas, situándose la primera (humedad relativa del aire) entre el 45 % y el 65 % y la segunda entre 23°C y 26°C en verano y entre 20°C y 24°C en invierno.

- Corrientes de aire molestas o intensas.
- Olores desagradables o que puedan causar mareos.

Iluminación

La iluminación del lugar de trabajo debe adaptarse a la naturaleza del mismo. Al ser este un trabajo de oficina, se deben tener en cuenta los riegos para la salud de los trabajadores y una posible disminución de su rendimiento que pudiera producirse por unas malas condiciones de visibilidad para cada una de las diferentes tareas a desarrollar, siendo necesario elegir el correcto tipo de iluminación:

- Se primará la utilización de iluminación natural.
- Como complemento a la anterior se podrá utilizar iluminación artificial, preferiblemente con capacidad de regulación a lo largo del día.
- Iluminación localizada para zonas concretas que así lo requieran.
- Evitar distribuciones de iluminación no uniformes así como deslumbramientos producidos por las fuentes de iluminación de forma directa o al reflejarse en una superficie.

Ruido

La exposición prolongada a un ruido de gran intensidad puede causar problemas auditivos permanentes. Por ello, los riesgos derivados de esta exposición deben eliminarse o reducirse, evitando mantener una intensidad continuada mayor a 55 dB utilizando equipos con emisiones sonoras mínimas y adaptando acústicamente si es necesario el lugar de trabajo. Además, las zonas con niveles de ruido perjudiciales deberán estar debidamente señalizadas, teniendo derecho a utilizar dispositivos auditivos protectores.

Salidas de emergencia y protección contra incendios

El área de trabajo debe contar con medidas en caso de emergencia tales como vías de evacuación, que deben estar correctamente señalizadas y cuya existencia y protocolos de utilización deben darse a conocer a los trabajadores para su correcto uso. Además, dependiendo de la capacidad y naturaleza del lugar de trabajo, instalaciones contra incendios tales como puertas cortafuegos o extintores deberán estar presentes, en buen estado y señalizados.

Instalación eléctrica

Debido a la presencia de equipos electrónicos se debe reducir el nivel de radiación electromagnética y garantizar el mantenimiento de cables y conexiones. Tanto este último como la instalación eléctrica deberá ser mantenida de manera periódica por una empresa autorizada y monitorizada para la detección de desperfectos que pudieran originar incendios o explosiones.

Ergonomía

Se debe asegurar mediante el uso del mobiliario correspondiente que la postura del trabajador en jornadas prolongadas es la correcta, pues de lo contrario le pueden ocasionar problemas de salud importantes. Este mobiliario incluye pantallas, sillas o escritorios, que deben tener el tamaño, diseño o distancia correctos para asegurar la correcta postura del trabajador.

8.3. Etapas del proyecto

Se detallan en esta sección las diferentes etapas presentes en el desarrollo del proyecto, de forma que sea así más sencillo entender posteriormente las horas dedicadas a cada etapa y por tanto el coste monetario proveniente de ellas.

1. **Aprendizaje de las herramientas computacionales**: incluye el periodo inicial de aprendizaje de las herramientas computacionales usadas en la realización del proyecto como la sintaxis del lenguaje de programación FORTRAN, los conceptos generales de computación paralela y la implementación de sus estándares OpenMP y MPI. Se llevó a cabo mediante lectura de bibliografía relacionada y ejercicios propuestos por el tutor.

2. **Revisión de conceptos de turbulencia**: incluye el periodo de revisión bibliográfica del estado del arte en las simulaciones DNS dedicadas al estudio de la turbulencia. Se llevó a cabo mediante la lectura de artículos científicos publicados y libros relacionados con la materia.

3. **Familiarización con el código provisto**: incluye el periodo de estudio y comprensión de la implementación y funcionamiento del código proporcionado por el tutor. Se llevó a cabo mediante ejercicios propuestos por el tutor. Ha sido la etapa más complicada.

4. **Desarrollo de los algoritmos**: incluye el periodo de realización de los cambios en el código y el desarrollo de algoritmos propuestos como mejora. Se ha llevado a cabo de forma autónoma siguiendo unas guías básicas dictadas por el tutor.

5. **Generación de resultados**: incluye el periodo de revisión de los cambios implementados en el código y el proceso de evaluación de los mismos en términos de rendimiento mediante su comparación.

6. **Redacción de la memoria**: incluye el periodo final de redacción de la memoria del proyecto, donde se detalla el trabajo realizado, los conceptos necesarios para comprenderlo de forma correcta y se presentan los resultados obtenidos analizados.

8.4. Recursos materiales

Para el desarrollo del proyecto se han utilizado recursos materiales tanto físicos (*hardware*), donde se incluyen los ordenadores, como virtuales (*software*), donde se incluyen las licencias de los programas informáticos utilizados. A continuación se describen de manera breve todos los materiales utilizados cuyos costes se detallarán en el capítulo siguiente.

Hardware

- **Supercomputador** - MareNostrum IV [28]
  - **Procesador**: 2x Intel Xeon Platinum 8160 a 2,1 GHz con 24 núcleos y 48 hilos
  - **Memoria RAM**: de 96 a 384 GB DDR4 a 2666 MHz por nodo
  - **Red de conexión**: 100Gb Intel Omni-Path y 10Gb Ethernet

- **Supercomputador** - SuperMUC-NG [29]
  - **Procesador**: 2x Intel Xeon Platinum 8174 a 3,1 GHz con 24 núcleos y 48 hilos
  - **Memoria RAM**: de 96 a 768 GB DDR4 a 2666 MHz por nodo
  - **Red de conexión**: 100Gb Intel Omni-Path

- **Ordenador portátil** - HP Pavilion Power 15
  - **Procesador**: Intel Core i7-7700HQ a 2,8 GHz con 4 núcleos y 8 hilos
  - **Memoria RAM**: 8 GB DDR4 a 2400 MHz
  - **Almacenamiento**: Disco SSD de 0.5 TB

- **Ordenador de sobremesa** - Personalizado
  - **Procesador**: AMD FX 8350 a 4 GHz con 8 núcleos y 8 hilos
  - **Memoria RAM**: 16 GB DDR3 a 1600 MHz
  - **Almacenamiento**: Disco HDD de 1 TB y SSD de 0.5 TB

Software

- **Compilador FORTRAN**: Se han utilizado tanto el compilador gfortran (libre) como el ifort de Intel (privativo, comercial) incluido en su paquete de software Intel Parallel Studio XE para compilar el código desarrollado en FORTRAN. Ambos incluyen el estándar OpenMP y el paquete de Intel incluye también una versión propia del estándar MPI.
- **HDF5** (libre): Utilizado como interfaz de lectura y escritura de archivos de datos.
- **FFTW3** (libre): Utilizado para las operaciones de las transformadas discretas de Fourier en el código.
- **MATLAB** (privativo, comercial): Utilizado para el pos-procesado de datos y la generación de gráficos.
- **Eclipse** (libre): Utilizado como IDE durante el desarrollo de los algoritmos.
- **LaTeX** (libre): Utilizado como procesador de textos para la redacción de la memoria del proyecto.

Capítulo 9

Presupuesto

Se hace necesario en cualquier proyecto de ingeniería elaborar un presupuesto que recoja todos los gastos que ha supuesto su elaboración. Para ello se consideran los distintos factores que han intervenido, como el coste monetario por tipo de persona cualificada (ingeniero o ingeniero superior/doctor) dependiendo de las horas trabajadas por cada uno o el coste del material utilizado, ya sea a nivel de *hardware* o *software*. La unidad monetaria será el euro (€).

9.1. Costes humanos

Para poder contabilizar el coste humano es necesario establecer un coste monetario que refleje el tiempo (en horas) de dedicación al proyecto y que dependa de la cualificación del trabajador. Así, se establece un coste aproximado de 15 €/h para el ingeniero y de 30 €/h para el tutor, ya sea ingeniero superior o doctor. El desglose de costes relacionados con las tareas desempeñadas puede observarse en la Tabla 9.1.

9.2. Costes materiales

También se hace necesario contabilizar el coste de los recursos materiales utilizados en la realización del proyecto. Se pueden distinguir los costes de los equipos físicos, cuyo valor se estima como la parte proporcional de su uso durante el desarrollo del trabajo (6 meses) considerando un tiempo total de vida de unos 5 años de uso, y los costes derivados de la utilización de licencias de programas informáticos. El desglose de costes relacionados con los materiales puede observarse en la Tabla 9.2, en los que se han incluido los costes derivados del uso de los dos supercomputadores a un coste de 0.01 € por CPU/h.

9.3. Coste total

Por último, a los costes en bruto producidos por los recursos materiales y humanos se les debe añadir un 21 % de IVA. El desglose de los costes finales y el coste total del proyecto puede observarse en la Tabla 9.3.

Categoría	Concepto	Horas	Coste	Subtotal
Etapa 1	Ingeniero	100	15 €/h	1500 €
	Doctor	10	30 €/h	300 €
Etapa 2	Ingeniero	30	15 €/h	375 €
	Doctor	5	30 €/h	150 €
Etapa 3	Ingeniero	100	15 €/h	1500 €
	Doctor	10	30 €/h	300 €
Etapa 4	Ingeniero	150	15 €/h	2250 €
	Doctor	5	30 €/h	150 €
Etapa 5	Ingeniero	60	15 €/h	900 €
	Doctor	30	30 €/h	900 €
Etapa 6	Ingeniero	100	15 €/h	1500 €
	Doctor	10	30 €/h	300 €
Total	Ingeniero	540	15 €/h	8100 €
	Doctor	70	30 €/h	2100 €
			Total	**10200 €**

Tabla 9.1: Coste humano por horas de trabajo.

Categoría	Concepto	Subtotal
Hardware	Portátil	75 €
	Sobremesa	95 €
	Tiempo en supercomputadores	250 €
Software	Intel Parallel Studio	1325 €
	Licencia de Matlab	800 €
	Total	**2545 €**

Tabla 9.2: Coste material.

Concepto	Impuesto	Subtotal
Horas de Trabajo	-	10200 €
Recursos Materiales	-	2545 €
IVA	21 %	2676 €
	Total	**15421 €**

Tabla 9.3: Coste total del proyecto.

El coste total del proyecto asciende a:

QUINCE MIL CUATROCIENTOS VEINTIUNO

Bibliografía

[1] J. Carlson, A. Jaffe, A. Wiles, and Clay Mathematics Institute. *The Millennium Prize Problems*. American Mathematical Society, 2006.

[2] S. Hoyas and J. Jiménez. Scaling of the velocity fluctuations in turbulent channels up to Re_τ=2003. *Physics of fluids 18*, 2006.

[3] S. Hoyas, M. Oberlack, S. Kraheberger, and F. Alcantara-Avila. Turbulent channel flow at Re_τ=10000. *American Physical Society*, 2019.

[4] NASA Juno Mission, available at https://www.nasa.gov/juno/, accessed June 2020.

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

[6] O. Reynolds. An experimental investigation of the circumstances which determine whether the motion of water shall be direct or sinuous, and of the law of resistance in parallel channels. *Philosophical Transactions of the Royal society of London 174*, 1883.

[7] Y. Çengel. *Mecánica de fluidos*. McGraw-Hill S.L., 2006.

[8] S.B. Pope. *Turbulent Flows*. Cambridge University Press, 2000.

[9] A.N. Kolmogorov. The local structure of turbulence in incompressible viscous fluid for very large reynolds numbers. *C.R. Acad. Sci. U.R.S.S. 30*, 1941.

[10] P. Davidson. *Turbulence: An Introduction for Scientists and Engineers*. Oxford University Press, 2015.

[11] J. Jiménez. Computers and turbulence. *European Journal of Mechanics-B/Fluids 79*, 2020.

[12] J. Hart. Comparison of turbulence modeling approaches to the simulation of a dimpled sphere. *Procedia engineering*, 147:68–73, 2016.

[13] S.A. Orszag and G.S. Patterson. Numerical simulation of three-dimensional homogeneous isotropic turbulence. *Physical Review Letters 28*, 1972.

[14] J. Jiménez. *Turbulence and vortex dynamics*. 2004.

[15] J. Kim, P. Moin, and R. Moser. Turbulence statistics in fully developed channel flow at low reynolds number. *Journal of fluid mechanics 177*, 1987.

[16] F. Lluesma-Rodríguez, S. Hoyas, and M.J. Perez-Quiles. Influence of the computational domain on dns of turbulent heat transfer up to Re_τ=2000 for Pr= 0.71. *International Journal of Heat and Mass Transfer 122*, 2018.

[17] C. Canuto, M.Y. Hussaini, A. Quarteroni, A. Thomas Jr, et al. *Spectral methods in fluid dynamics*. Springer Science & Business Media, 2012.

[18] S.K. Lele. Compact finite difference schemes with spectral-like resolution. *Journal of computational physics 103*, 1992.

[19] A. Pallarés. *Herramientas numéricas para la resolución de ecuaciones diferenciales*. Universitat Politécnica de València, 2004.

[20] P.R. Spalart, R.D. Moser, and M.M. Rogers. Spectral methods for the navier-stokes equations with one infinite and two periodic directions. *Journal of Computational Physics 96*, 1991.

[21] M. Lee and R.D. Moser. Direct numerical simulation of turbulent channel flow up to Re_τ=5200. *Journal of Fluid Mechanics 774*, 2015.

[22] Top500 - The list, available at https://www.top500.org, accessed June 2020.

[23] G. Hager and G. Wellein. *Introduction to high performance computing for scientists and engineers*. CRC Press, 2010.

[24] OpenMP, available at https://www.openmp.org/, accessed June 2020.

[25] MPI, available at https://www.mpich.org or *https://www.open-mpi.org/*, accessed June 2020.

[26] Fastest Fourier Transform in the West, available at http://www.fftw.org/, accessed June 2020.

[27] TheHDFGroup, available at https://www.hdfgroup.org/, accessed June 2020.

[28] Documentación de hardware de MareNostrum, available at https://www.bsc.es/marenostrum/marenostrum/technical-information/, accessed June 2020.

[29] Documentación de hardware de SuperMUC, available at https://doku.lrz.de/display/PUBLIC/Hardware+of+SuperMUC-NG/, accessed June 2020.

[30] I. Querol. *Estudio DNS de un problema de estratificación en la mesosfera*. Universitat Politécnica de València, 2019.

Parte III

Anexos

TOP 500
The List.

PRESENTED BY

ICL INNOVATIVE COMPUTING LABORATORY

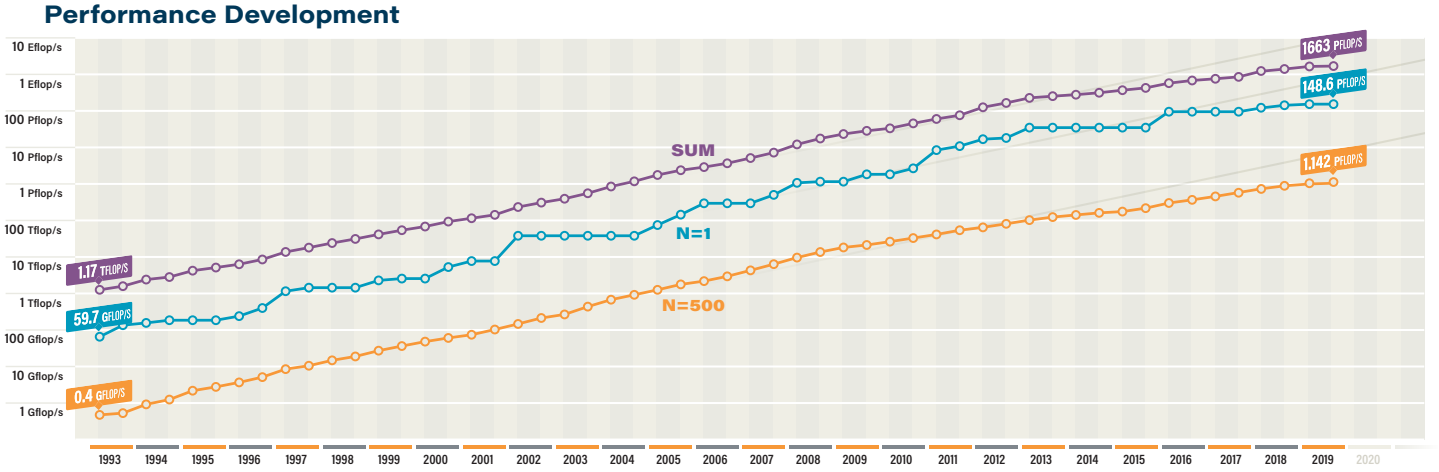
BERKELEY LAB Lawrence Berkeley National Laboratory

ISC GROUP Moving Forward.

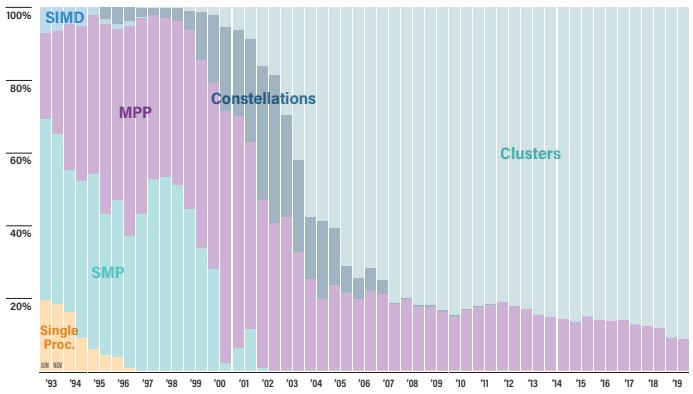
FIND OUT MORE AT
top500.org

	SYSTEM	SPECS	SITE	COUNTRY	CORES	RMAX PFLOP/S	POWER MW
1	**Summit**	IBM POWER9 (22C, 3.07GHz), NVIDIA Volta GV100 (80C), Dual-Rail Mellanox EDR Infiniband	DOE/SC/ORNL	USA	2,414,592	148.6	11.4
2	**Sierra**	IBM POWER9 (22C, 3.1GHz), NVIDIA Tesla V100 (80C), Dual-Rail Mellanox EDR Infiniband	DOE/NNSA/LLNL	USA	1,572,480	94.6	7.44
3	**Sunway TaihuLight**	Shenwei SW26010 (260C, 1.45 GHz) Custom Interconnect	NSCC in Wuxi	China	10,649,600	93.0	15.4
4	**Tianhe-2A (Milkyway-2A)**	Intel Ivy Bridge (12C, 2.2 GHz) & TH Express-2, Matrix-2000	NSCC Guangzhou	China	4,981,760	61.4	18.5
5	**Frontera**	Dell C6420, Xeon Platinum 8280 28C 2.7GHz, Mellanox InfiniBand HDR	TACC/U of Texas	USA	448,448	23.5	-

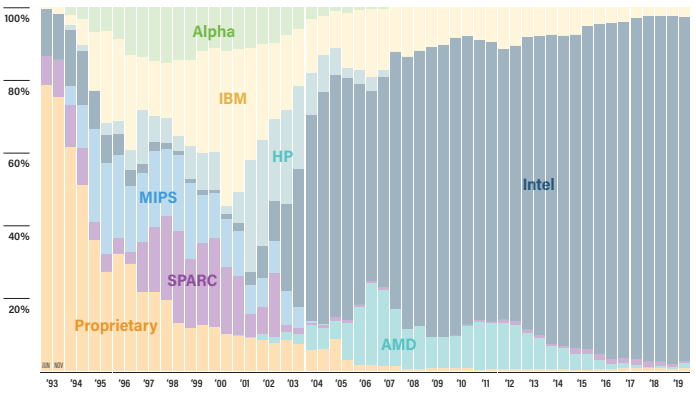
Performance Development



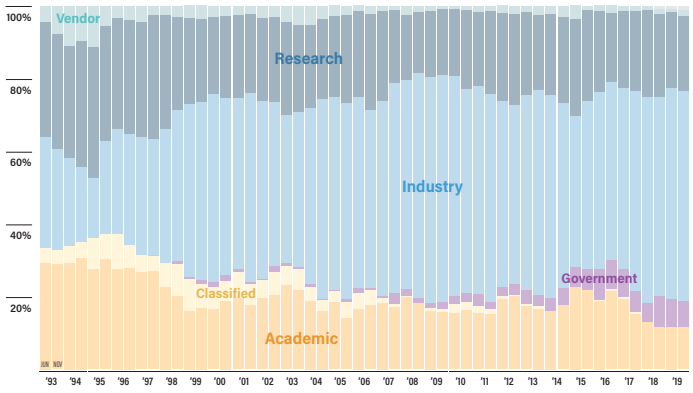
Architectures



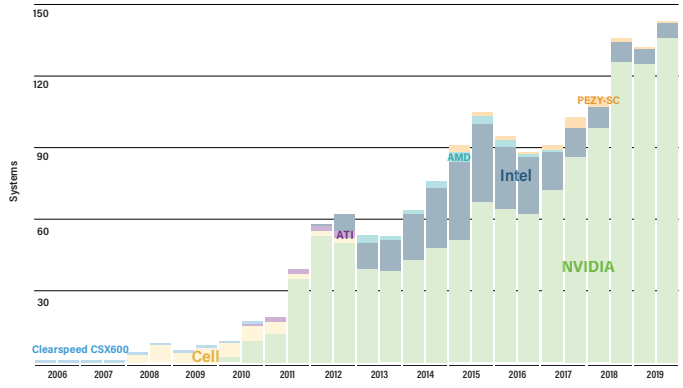
Chip Technology



Installation Type



Accelerators/Co-processors



HPLINPACK *A Portable Implementation of the High Performance Linpack Benchmark for Distributed Memory Computers* ▸ MORE INFO AT http://icl.utk.edu/hpl/