Capítulo 6

Implementación del código numérico

En este capítulo se va a describir las características de la simulación y de la implementación del código así como los puntos donde se puede aplicar la paralelización de los cálculos.

6.1. Introducción

El programa divide el cubo en una serie de puntos equiespaciados en forma de mayado ortogonal en los que se realizan los cálculos. En cada punto del cubo se define el vector de velocidades con sus tres componentes y la parte principal del programa se encarga de controlar la evolución temporal del campo de velocidades en el cubo. Para realizar esta evolución, recurrimos a modelar la dinámica del sistema con las ecuaciones de Navier-Stokes (ecuación 4.1) junto a la imposición de que el cubo no tiene ni fuentes ni sumideros, o lo que es lo mismo, que su divergencia es cero (ecuación 4.2). En el capítulo 4 se detallan las expresiones que conducen a la ecuación que dado un estado nos permite obtener el siguiente (ecuaciones 4.42 y 4.48). El objetivo de ese capítulo es encontrar un método que sea eficiente computacionalmente hablando.

En una transformada DFT como la que usamos en este proyecto, obtenemos términos en frecuencia que se corresponden con valores de $|\mathbf{k}|$ que van desde 0 en el centro hasta $\sqrt{\frac{3}{4}}N$ en las esquinas del cubo. Debido a los errores numéricos los términos correspondientes a valores de $|\mathbf{k}|$ altos tienen una gran componente de ruido por los fenómenos de aliasing y truncamiento de la serie, y no son de interés para esta simulación. Por esto, al igual que en el artículo de L. P. Wang y M. R. Maxey [23], se aplica un filtro esférico que elimina las componentes en frecuencia del campo de velocidades en las que $|\mathbf{k}| > \frac{N}{2} - 1{,}5$.

El forzado en esta simulación se va a realizar introduciendo el concepto de viscosidad negativa en las componentes espectrales de módulo menor que $2,25$; es decir, en los cálculos que se realizan para obtener el siguiente estado, sustituimos la viscosidad de los términos en los que $|\mathbf{k}| < 2{,}5$ por cierto valor negativo. Este valor se calcula mediante un PID [16] de forma que se mantenga un estado estacionario en el que la energía que se aporta al sistema sea igual a la que se disipa. Como variable de control del PID se elige que el producto $K_{max}\eta \simeq \frac{N}{2}\eta$ sea un valor constante e igual a 1,4 (aunque este valor es configurable).

De los artículos publicados sobre esta materia, se sabe que para estas simulaciones el paso de tiempo óptimo es $dt = 0{,}025T_\eta = 0{,}025\left(\frac{\nu}{\epsilon}\right)^{\frac{1}{2}}$, por eso el código mantiene el paso en el intervalo $0{,}015 < \frac{dt}{T_\eta} < 0{,}025$.

6.2. Codificación

El código se ha programado en lenguaje C [13] que junto con el Fortran son los lenguajes más comunes en la programación de simulaciones numéricas. En la implementación de las distintas características se van a usar tres librerías que nos facilita la codificación.

Librería FFTW

Como comentamos en la sección 3.2.5, esta librería implementa la transformada FFT. Utilizamos esta librería por ser una de las implementaciones del algoritmo más eficientes. Esta eficiencia se basa en el uso de distintas subrutinas específicas para el procesador en el que se compila así como en un análisis previo de las mejores opciones en cuanto al tamaño de la segmentación y otros parámetros internos para obtener la máxima velocidad posible.

Librería NMPI

Como comentamos en la sección 5.3 esta librería implementa el interfaz de paso de mensajes MPI, está basada en el paquete MPICH2 con ciertas modificaciones para usar las tarjetas SCI. Esta librería es con la que llevamos a cabo la paralelización del código, no sólo por las funciones que nos permiten sincronizar la ejecución en los nodos y el intercambio de información, sino porque también se integran con la librería FFTW descargando de la gran complejidad que supone hacer un código optimizado para calcular una FFT en paralelo.

Librería ImageMagick

Si bien esta librería no es necesaria para la simulación, sí nos permite obtener resultados gráficos de la misma. Esta librería se usa para generar las imágenes del espectro, los invariantes y la evolución de $\frac{N}{2}\eta$ en las rutinas de resultados. La ventaja de realizar las gráficas con el programa son que además de dar información en tiempo real del estado de la simulación, en el caso de los invariantes, nos permite calcular la gráfica utilizando el cluster lo cual debido al enorme tamaño del archivo de invariantes es casi necesario.

6.3. Diagramas de flujo

El programa principal tiene el diagrama de flujo de la figura 6.1. Como se puede ver, hay tres bloques:

- El primero es en el que se preparan las condiciones iniciales. Se inicializa el entorno MPI, se reserva la memoria suficiente para definir el estado, se inicializa, y se da el primer paso con el método de Runge-Kutta de segundo orden.
- En el segundo es en el que realizamos la evolución temporal así como la extracción de datos y parámetros en estudio.
- En el tercero se guarda el estado para un futuro uso, y se liberan los recursos del cluster.

En la figura 6.2 se puede ver en detalle el diagrama de flujo del bloque principal. A continuación vamos a describir las características más destacadas de la implementación.

El algoritmo de Adams-Bashforth descrito por la ecuación 4.48 al ser de segundo orden, necesita de las variables en dos instantes para calcular el siguiente, por tanto la memoria que necesitamos reservar es:

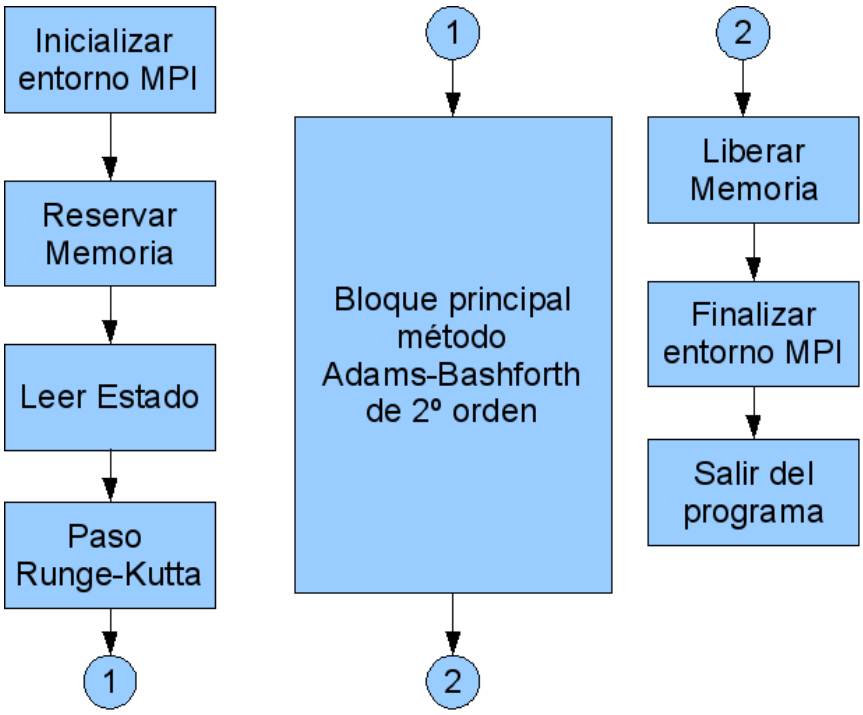

Figura 6.1: Diagrama de flujo.

- Estado actual:
  - Vector de velocidades: $\widehat{\mathbf{v}}_k^{t_n}$ Memoria: $3 * N^3$
  - Término no lineal: $\left(\widehat{\mathbf{v} \times \omega}\right)_k^{t_n}$ Memoria: $3 * N^3$
  - Espectro: Memoria: $N/2$
  - Matriz de invariantes: Memoria: $9 * N^3$
  - Cálculo de imágenes: $640 * 480$ de fondos + 1000 de gráfica de evolución del $K\eta$
- Estado anterior: Memoria: $3 * N^3$
  - Término disipativo: $\left(\widehat{\mathbf{v} \times \omega}\right)_k^{t_{n-1}}$ Memoria: $3 * N^3$
- Nuevo estado:
  - Vector de velocidades: $\widehat{\mathbf{v}}_k^{t_{n+1}}$ Memoria: $3 * N^3$. Este vector también se usa como variable temporal durante el cálculo para ahorrar memoria.

Es decir, el orden de las necesidades de memoria es $21 * N^3$ números reales (en precisión simple para $N = 64$ son aproximadamente 22 MBytes, para $N = 128$ son aproximadamente 176 MBytes,...). La memoria necesaria para las matrices de tamaño N^3 la dividimos entre los nodos del cluster, para ello, usamos la función *rfftwnd_mpi_local_sizes* de la librería fftw que nos permite obtener los valores de offset y tamaño de la matriz local a cada nodo.

Por motivos de eficiencia computacional, la librería fftw la vamos a configurar de forma que la transformada de Fourier calcula la matriz transpuesta de la que queremos que calcular,

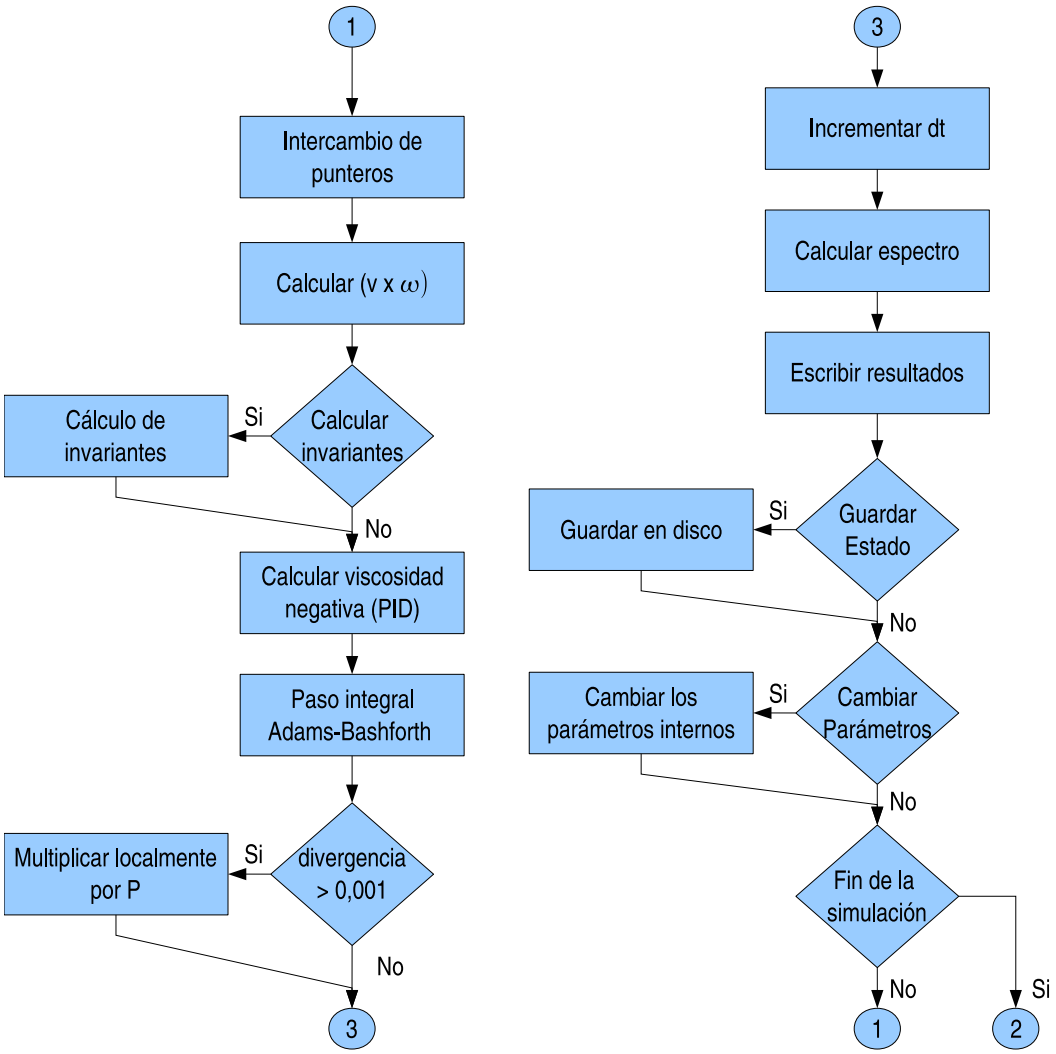

Figura 6.2: Detalle del diagrama de flujo.

mediante esta pequeña complicación en el código, ahorramos una gran cantidad de intercambios de bloques de memoria entre los nodos. Se puede ver un pequeño ejemplo del uso de la librería fftw en el cuadro de código 6.1.

Los pasos a seguir para realizar una transformada con la librería fftw son

1. Crear los planes de ejecución con la función *rfftw3d_mpi_create_plan*.

2. Calcular el tamaño de la memoria necesaria en cada nodo y los valores de offset respecto a la original con la función *rfftwnd_mpi_local_sizes*.

3. Reservar la memoria necesaria, para ahorrar memoria, en las transformadas de valores reales, la librería utiliza la mitad de la memoria, pero si el número de muestras es par se usa un poco más de memoria, dada una secuencia de tamaño N, la cantidad de memoria que se necesita es [2*(N/2+1)].

4. Inicializar los datos.

5. Transformar con la función *rfftwnd_mpi*.

6. Usar los datos.

7. Liberar la memoria.

8. Destruir los planes con *rfftwnd_mpi_destroy_plan*.

```
// Inicializamos el entorno MPI
MPI_Init(&argc,&argv);
//Creamos los planes
Plan_mpi = rfftw3d_mpi_create_plan(MPI_COMM_WORLD, N, N, N,
    FFTW_REAL_TO_COMPLEX, FFTW_ESTIMATE);
PlanInv_mpi = rfftw3d_mpi_create_plan(MPI_COMM_WORLD, N, N, N,
    FFTW_COMPLEX_TO_REAL, FFTW_ESTIMATE);
//Obtenemos los tamaños de las matrices en el nodo
rfftwnd_mpi_local_sizes(Plan_mpi, &TAM_LOCAL.NX, &TAM_LOCAL.
    NX_offset, &TAM_LOCAL.NY, &TAM_LOCAL.NY_offset, &TAM_LOCAL.
    TOTAL);
//Reservamos memoria
VX.Esp = fftw_malloc(sizeof(fftw_real) * TAM_LOCAL.TOTAL);
//Espacio de memoria auxiliar para aumentar la velocidad
EspacioAux.Esp = fftw_malloc(sizeof(fftw_real) * TAM_LOCAL.TOTAL
    );
//Inicialización de los datos (campo aleatorio)
srand(MPI.PROCESO_MPI);
for(x=0;x<TAM_LOCAL.NX;x++)
    for(y=0;y<N;y++)
        for(z=0;z<N;z++)
        {
            if(rand()>RAND_MAX/2.0)
                VX.Esp[z+(2*((N/2)+1))*(y+N*x)]=AMPLITUD_INICIAL*(
                    (1.0*rand())/(RAND_MAX+1.0));
            else
                VX.Esp[z+(2*((N/2)+1))*(y+N*x)]=-AMPLITUD_INICIAL*(
                    (1.0*rand())/(RAND_MAX+1.0));
        }
//Ejemplo de transformada directa
rfftwnd_mpi(Plan_mpi, 1, VX.Esp, EspacioAux.Esp,
    FFTW_TRANSPOSED_ORDER);
//Ejemplo de transformada inversa
rfftwnd_mpi(PlanInv_mpi, 1, VX.Esp, EspacioAux.Esp,
    FFTW_TRANSPOSED_ORDER);
//Liberarmos la memoria
free(VX.Esp);
free(EspacioAux.Esp);
rfftwnd_mpi_destroy_plan(Plan);
rfftwnd_mpi_destroy_plan(PlanInv);
//Terminamos el entorno MPI
MPI_Finalize();
```

Código fuente 6.1: Ejemplo de transformada con fftw.

El programa utiliza dos archivos de parámetros para gestionar el cambio en tiempo real de los parámetros de la simulación (ver apartado 6.4) como la viscosidad, el tamaño del filtro, etc.

Dado la gran cantidad de datos numéricos que son los resultados de este programa, se ha optado porque además de los archivos con los valores, el programa genere un resultado gráfico que ayude a analizar los mismos. Para crear las imágenes se utiliza la librería ImageMagick que nos permite realizar distintos tipos de gráficas. Un breve tutorial de su uso sería el siguiente:

1. Reservar un espacio en memoria del tamaño de la imagen.
2. Crear el objeto MagickWand que es el que genera la imagen.
3. Crear el objeto PixelWand que nos permite seleccionar las características de estilo de los objetos que se dibujan.
4. Crear el objeto DrawingWand que hace de interfaz de dibujo en el objeto MagickWand usando las características de PixelWand.
5. Usar las funciones de dibujo que permiten modificar las características del PixelWand, por ejemplo *PixelSetColor*.
6. Usar las funciones de dibujo que permiten pintar en el dibujo, por ejemplo *DrawLine*.
7. Crear la imagen en disco con *MagickWriteImages*.
8. Destruir los objetos DrawingWand, PixelWand y MagickWand.

6.4. Configuración en tiempo real

Como se ha comentado, el programa admite el ajuste de sus parámetros internos en tiempo real, para lo cual se utilizan dos archivos

- Archivo de entrada de parámetros: *Parametros.dat*. En este archivo se le pasan las indicaciones al programa para que modifique alguno de los parámetros internos.
- Archivo de indicación de parámetros: *Parametros-salida.dat*. En este archivo el programa va indicando los valores de los parámetros en cada paso de integración.

El formato de los parámetros es el mismo en ambos archivos, son archivos de texto en los que cada línea indica un parámetro, se van leyendo consecutivamente por lo que si se repite alguno el que tiene efecto es el último. Los parámetros que se pueden pasar son:

- FORZADO: Indica si se usa el forzado para inyectar energía al sistema o se deja la evolución libre.
- CALCULAINVARIANTES: Indica si se deben calcular los invariantes.
- GUARDAESTADO: Indica si se debe guardar el estado en disco.
- Pasos: Indica cuántos pasos integrales debe dar el programa.
- VISCOSIDAD: Valor de la viscosidad.

- dt: Valor del siguiente paso de integración. Si se usa este parámetro, el siguiente paso integral se da usando el método de Runge-Kutta con el nuevo valor.
- K_FILTRO: Valor del $|k|$ de los mayores modos que se calculan. Los modos que tengan un $|k|$ mayor se fuerza a que valgan cero.
- CURSOR_K: Indica dónde se pone el cursor vertical que sirve para medir sobre la gráfica del espectro.
- CURSOR_E: Indica dónde se pone el cursor horizontal que sirve para medir sobre la gráfica del espectro.
- KETA_ESTACIONARIA: Indica el valor de $k_{max}\eta$ al que el PID intenta estabilizar el sistema.
- TIEMPOTOTAL: Indica el tiempo total transcurrido de integración.
- PID_K_P: Valor de la constante proporcional del PID.
- PID_K_I: Valor de la constante integral del PID.
- PID_K_D: Valor de la constante derivativa del PID.
- PID_error: Valor del error en el paso de integración que se usa junto con la constante proporcional en PID.
- PID_deriv_error: Valor de la derivada del error en el paso de integración que se usa junto con la constante derivativa en PID.
- PID_int_error: Valor de la integral del error en el paso de integración que se usa junto con la constante integral en PID.
- Imagen_Ancho: Ancho de las imágenes que se generan.
- Imagen_Alto: Alto de las imágenes que se generan.
- Imagen_KETA_MAX: Valor de $k_{max}\eta$ máximo que aparece en la gráfica de $k_{max}\eta$.
- Imagen_KETA_MIN: Valor de $k_{max}\eta$ mínimo que aparece en la gráfica de $k_{max}\eta$.
- Escala_Invariantes_X: Factor por el que se multiplican los valores del determinante en la gráfica de los invariantes.
- Escala_Invariantes_Y: Factor por el que se multiplican los valores del invariante asociado a los adjuntos en la gráfica de los invariantes.
- Imagen_Divisiones_X: Número de décadas en la gráfica del espectro.
- Imagen_Divisiones_Y: Número de divisiones en el eje Y en la gráfica del espectro.

Para poder gestionar todos estos parámetros se ha realizado un pequeño programa que lee los parámetros de salida y nos permite ajustar los parámetros y generar el archivo de paso de parámetros cómodamente. En la figura 6.3 se puede ver el funcionamiento del programa. El programa llamado *ajustanavier* ha sido diseñado para el entorno KDE bajo entornos Linux. Su uso es muy simple, se debe ejecutar en el mismo directorio que el programa principal y se sincroniza automáticamente. En realidad, el programa *ajustanavier* se suele ejecutar en un ordenador que no pertenece al cluster ya que este no dispone de entorno gráfico, lo que se hace es montar el directorio de trabajo del cluster en el ordenador del usuario mediante la red NFS y ejecutar el programa de configuración en el entorno gráfico del mismo.

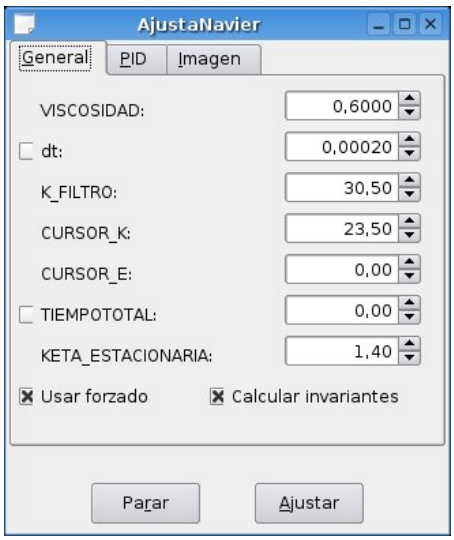

Figura 6.3: Programa de ajuste de parámetros internos en tiempo real.

6.5. Compilación y ejecución del programa

El código está estructurado en distintos archivos fuente, para compilar y crear el ejecutable a partir de los mismos ejecutamos una de las siguientes órdenes, la diferencia entre ambas es que la primera genera un código con precisión doble y el segundo con precisión simple.

```
mpicc -o navier_mpi_doble -lm -lfftw -lrfftw -lMagick -lWand
   -lrfftw_mpi -lfftw_mpi *.c
```

Código fuente 6.2: Generación del ejecutable con precisión doble a partir del código fuente.

```
mpicc -o navier_mpi_simple -DPrecisionSimple -lm -lsfftw -
    lsrfftw
   -lMagick -lWand  -lsrfftw_mpi -lsfftw_mpi -lfftw -lfftw_mpi -
       lrfftw
   -lrfftw_mpi *.c
```

Código fuente 6.3: Generación del ejecutable con precisión simple a partir del código fuente.

6.6. Paralelización

6.6.1. Introducción

El programa que vamos a implementar tiene bastantes cálculos que se pueden realizar de forma paralela, se puede dividir la forma en la que se va a abordar la paralelización en tres tipos de casos.

Intercambio de parámetros: En este tipo de casos, como puede ser el paso de los valores para iniciar una nueva simulación (N, viscosidad, etc.), se usan las funciones adecuadas del interfaz MPI.

Cálculos en cada punto: En este tipo de casos, como pueden ser los pasos integrales, los cálculos únicamente necesitan de las propiedades de cada punto del cubo, en este caso cada nodo opera sobre los puntos del cubo que tiene en su memoria local. No es necesario ningún tipo de sincronización entre los nodos.

Cálculos de suma o media en todos los puntos del cubo: En este tipo de casos, como puede ser el cálculo del espectro, cada nodo realiza las operaciones necesarias en los puntos de su memoria local y al final se sincroniza con los demás para realizar la suma o media total.

Cálculos en los que están involucrados todos los puntos del cubo: En este tipo de casos la paralelización resulta muy complicada, en nuestro código únicamente se presenta este tipo de casos en los cálculos de los coeficientes de las series de Fourier, y de su correcta y eficiente resolución se ocupa la librería fftw.

6.6.2. Uso de índices

Como ya hemos comentado, las librerías fftw se encargan del cálculo de la FFT usando MPI internamente. Por motivos de eficiencia computacional, las matrices tridimensionales en frecuencia se calculan traspuestas, por tanto, teniendo en cuenta que los bucles están traspuestos y el distinto tamaño de las matrices en los dos dominios, los bucles para recorrer las matrices son:

En el espacio físico

Las variables x, y, z son los índices que representan la posición del punto en el cubo.

```
for(x=0;x<TAM_LOCAL.NX;x++)
{
    for(y=0;y<N;y++)
    {
        for(z=0;z<N;z++)
        {
            indice_Matriz=z+(2*((N/2)+1))*(y+N*x);
            .
            .
            .
```

Código fuente 6.4: Ejemplo de recorrido de matrices en el espacio físico.

En el espacio de Fourier

Las variables x, y, z sirven para desplazarse por el cubo en memoria, y las variables indice_x, indice_y e indice_z se corresponden con k_x, k_y y k_z.

```
for(y=0;y<TAM_LOCAL.NY;y++)
{
    indice_y=((y+TAM_LOCAL.NY_offset)>=(N/2))?((y+TAM_LOCAL.
        NY_offset)-N):(y+TAM_LOCAL.NY_offset);
    for(x=0;x<N;x++)
    {
        indice_x=(x>=(N/2))?(x-N):x;
        for(z=0;z<((N/2)+1);z++)
        {
            indice_z=(z>=(N/2))?(z-N):z;
            indice_Matriz=z+((N/2)+1)*(x+N*y);

            modulo_cuadrado_k=indice_z*indice_z+indice_x*indice_x+
                indice_y*indice_y;
            .
            .
            .
```

Código fuente 6.5: Ejemplo de recorrido de matrices en el espacio de Fourier.

6.6.3. Uso de la librería MPI

La librería se utiliza para la sincronización de los cálculos entre los distintos nodos. En esta sección se va a describir su uso en el código. Lo primero que hay que hacer es la inicialización de entorno MPI. Para ello se utilizan unas funciones (ver código fuente 6.6) que además de inicializar el entorno guardan la información referente a los procesos en una estructura que luego se utiliza para ajustar las matrices locales de cada nodo.

```
if(MPI_Init(&argc, &argv)!=MPI_SUCCESS)
{
    printf("Error: No se puede inicializar el MPI\n"); fflush(
        stdout);
    exit(1);
}
if(MPI_Comm_size(MPI_COMM_WORLD, &MPI.NUMERO_DE_PROCESOS_MPI)!=
    MPI_SUCCESS)
{
    printf("Error: No se puede inicializar el MPI\n"); fflush(
        stdout);
    exit(1);
}
if(MPI_Comm_rank(MPI_COMM_WORLD, &MPI.PROCESO_MPI)!=MPI_SUCCESS)
{
    printf("Error: No se puede inicializar el MPI\n"); fflush(
        stdout);
    exit(1);
}
```

Código fuente 6.6: Ejemplo de inicialización del entorno MPI.

Cuando se desee finalizar el programa, hay que cerrar el entorno MPI con la función:

```
MPI_Finalize();
```

Código fuente 6.7: Ejemplo de cierre del entorno MPI.

Para sincronizar los nodos se puede utilizar la función *MPI_Barrier* que hace que todos los nodos esperen en ese punto del código a que los otros lleguen. Esta función se utiliza al principio de cada bloque de operaciones para que todos los nodos empiecen a la vez el bloque.

```
MPI_Barrier(MPI_COMM_WORLD);
```

Código fuente 6.8: Ejemplo de sincronización de nodos.

Existen situaciones en las que el nodo principal, que en nuestro programa es el número cero, debe comunicar algún parámetro al resto, para ello se utiliza la función *MPI_Bcast*, en el siguiente ejemplo el proceso cero lee el valor de la variable N de la entrada estándar y luego la difunde al resto.

```
if(MPI.PROCESO_MPI==0)
{
    printf("\nValor de N: ");
    z=scanf("%d",&N);
    if(z==0)
    {
        printf("\nValor incorrecto");
        exit(1);
    }
}
MPI_Barrier(MPI_COMM_WORLD);
MPI_Bcast ( &N, 1, MPI_INTEGER, 0, MPI_COMM_WORLD);
```

Código fuente 6.9: Ejemplo de difusión de datos a los nodos.

En los casos en los que hay que realizar una serie de cálculos sobre todos los puntos del cubo, lo que se hace es que cada nodo realiza los suyos sobre una zona de memoria temporal, y luego utiliza la función *MPI_Allreduce* en la que se indica el tipo de operación que se realiza sobre los datos. En el ejemplo de código fuente 6.10 se ve la llamada que se usa para calcular el espectro del sistema (ver sección 7.4); en la zona de memoria apuntada por *Espectro.Valores_Rodaja_MPI* cada nodo ha realizado los cálculos asociados a sus puntos, y lo que hace este código es sumar en cada elemento de la zona de memoria apuntada por *Espectro.Valores_Rodaja* los valores equivalentes en las zonas *Espectro.Valores_Rodaja_MPI* de todos los nodos.

```
MPI_Barrier(MPI_COMM_WORLD);
MPI_Allreduce ( Espectro.Valores_Rodaja_MPI,Espectro.
    Valores_Rodaja, Espectro.tam, MPI_DOUBLE,MPI_SUM,
    MPI_COMM_WORLD);
```

Código fuente 6.10: Ejemplo de sincronización de datos entre los nodos.

Por último, queda comentar la forma en la que se usan los datos en disco, para ello se utilizan funciones de manejo de ficheros que ofrece el interfaz MPI. En el código fuente 6.11 se puede ver un ejemplo de acceso al disco para leer un parámetro.

```
//Variable para manejar ficheros con MPI:
MPI_File DATOS_MPI;
//Sincronizamos los nodos
MPI_Barrier(MPI_COMM_WORLD);
//Abrimos el fichero
if(MPI_File_open (MPI_COMM_WORLD, Nombre, MPI_MODE_RDONLY,
   MPI_INFO_NULL, &DATOS_MPI)!=MPI_SUCCESS)
{
   if(MPI.PROCESO_MPI==0)
   {
      fprintf(stderr, "\nNo puedo abrir %s\n",Nombre);
   }
   exit(1);
}
//Leemos un parámetro
MPI_File_read_all(DATOS_MPI, &N,sizeof(unsigned int),
   MPI_CHARACTER, &MPI.Estado);
//Cerramos el fichero
MPI_File_close(&DATOS_MPI);
```

Código fuente 6.11: Ejemplo de manejo de ficheros con MPI.