Actualización de Código Fortran 77 a Fortran 90/95 con OpenMP

Fernando G. Tinetti* Mónica A. López Juan C. Labraga

Instituto de Investigación en Informática (III-LIDI), Facultad de Informática, UNLP[1]
Centro Nacional Patagónico (CENPAT), CONICET[2]
Departamento de Informática, Facultad de Ingeniería, Sede Pto. Madryn, UNPSJB[3]

Reporte Técnico HPCGCM-01-2008[4]
Julio 2008

Resumen

En este reporte técnico se presentan algunas de las ideas a utilizar para el trabajo con código Fortran 77 heredado (*legacy code*) normalmente en áreas de aplicaciones numéricas. Si bien el objetivo *inicial* se orienta a aprovechar las ventajas de expresividad y controles estáticos de Fortran 90/95, no se puede dejar de lado la mejora *en general* del código fuente heredado a partir de la aplicación de ideas y/o técnicas más asociadas a ingeniería de software que a cómputo numérico. Se debe destacar en este sentido, que la mayoría de (o *todos*) los programas en producción actualmente e implementados en Fortran 77 tienen mayor énfasis en la calidad numérica del resultado que en los métodos de ingeniería de software y/o de implementación con un lenguaje de programación en particular, como Fortran 77. Claramente, el objetivo no es recodificar o reimplementar programas heredados sino aplicar ideas que ya son ampliamente aceptadas en la medida de lo posible en el código heredado. Por otro lado, muchos de los programas en excritos en Fortran 77 se orientan al aprovechamiento o a la optimización de rendimiento de las computadoras vectoriales. En este reporte, la idea será optimizar para multiprocesadores, utilizando el estándar OpenMP, que ha sido incluido por la gran mayoría de los compiladores Fortran 90/95 actuales. Esta optimización para aprovechar el cómputo paralelo de los multiprocesadores tiene su *justificación* en lo que parece ser un estándar para las empresas de procesadores, con la inclusión de múltiples núcleos de/en cada procesador.

1. Introducción

En el contexto de optimización de código numérico de un proyecto de investigación, aprovechamos la disponibilidad de código fuente Fortran 77 con dos objetivos en mente:

1. Analizar la codificación actual del programa básicamente desde la perspectiva de "actualización" de código Fortran 77 a Fortran 90/95. En este sentido, la idea *inicial* es analizar por un lado la posibilidad de mejorar el programa en cuanto legibilidad y abstracción y por el otro el nivel de dificultad de esta recodificación.

2. Analizar la posibilidad de aplicación de OpenMP [3] [2] con el objetivo de reducir el tiempo de ejecución en multiprocesadores. Expresado de otra manera, sería la optimización o *adaptación* de código para computadoras de múltiples núcleos (*multicore*) y/o con SMP (Symmetric MultiProcessing).

También como parte de la introducción se incluye información del programa sobre el cual se trabaja para este reporte técnico, con el objetivo de dar el contexto para todo el resto del texto y la justificación inicial de algunas de las decisiones que se han tomado.

---

* Investigador Asistente CICPBA
[1] Universidad Nacional de La Plata
[2] Consejo Nacional de Investigaciones Científicas y Técnicas
[3] Universidad Nacional de La Patagonia San Juan Bosco
[4] HPCCGM: sigla de High Performance Computing on Global Climate Modelling

1.1. Algunos Datos/Información del Programa en Fortran 77

Inicialmente, se dan algunos datos de análisis estático del código fuente del programa, en parte sin entrar en detalles (solamente casi a nivel *estadístico*) y en parte mostrando algunas características de la distribución del programa en diferentes archivos/ficheros. Posteriormente, se dan algunas ideas de tiempo de ejecución específicamente a partir de la utilización de *profiling* con la herramienta `gprof`.

Datos de Análisis Estático del Programa

1) Tiene 296 archivos .f, la mayoría de ellos con una única rutina Fortran. Menos del 10 % de los archivos .f se utilizan *solamente* para la definición de áreas common y/o variables y constantes.
2) Tiene poco menos de 58000 líneas *de código* en los archivos .f, incluyendo los comentarios.
3) Todo el código respeta el formato fijo de línea de 72 caracteres de Fortran 77. Las expresiones que utilizan más caracteres se continúan en la siguiente línea siguiendo la convención de poner un & en la columna 6.
4) Aproximadamente el 25 % de líneas de los archivos .f son comentarios. Se sigue la convención de incluir una c o C en la columna 1 (esto último de acuerdo con el formato fijo de línea de Fortran 77).
5) Por razones de manejo de la representación de números en los archivos intermedios que se generan y utilizan, al menos inicialmente se debe compilar el programa con `ifort`, el compilador de Fortran 77/90/95 de uso libre de Intel. Este compilador es similar al `gfortran` y al `g95` dado que tiene la posibilidad de compilar código Fortran 77 y Fortran 90/95. Por otro lado, también reconoce y compila las directivas de OpenMP para cómputo paralelo en multiprocesadores. Las opciones de compilación utilizadas con `ifort` son similares a las que se pueden utilizar con `gfortran` y `g95`. En particular, las que se utilizan son:

```
-O3 -p
```

para que optimice al máximo posible y que genere información de *profiling* para el análisis de la ejecución con la herramienta `gprof`.

Datos Obtenidos de la Ejecución del Programa (vía `gprof`)

Dado que el trabajo con el programa actual se da en el contexto de optimización de programas numéricos, lo primero a hacer es un conjunto de experimentos ejecutando el programa con *profiling* (utilizando `gprof`), para elegir la o las subrutinas específicas con las cuales trabajar. Esta primera ejecución en las computadoras disponibles (basadas en Intel Xeon) dio como resultado una lista ordenada de rutinas a analizar/actualizar/optimizar. El criterio de ordenación es, justamente, dado por el peso de tiempo de ejecución dado en tiempo absoluto y en % de tiempo total de ejecución del programa por la herramienta `gprof`. Utilizando la información que provee `gprof` se tiene:

- La rutina que tiene mayor tiempo de ejecución se ejecuta durante el 13 % del tiempo total.
- Las dos rutinas que tienen mayor importancia en tiempo de ejecución tienen más de 850 líneas de código cada una. Entre las primeras seis rutinas de mayor importancia en tiempo de ejecución aparecen dos intrínsecas. A priori, estas dos funciones intrínsecas no son consideradas como *optimizables*.

Con el objetivo de comparar la diferencia de rendimiento desde la perspectiva de hardware utilizado, también se ejecutó el programa en PCs de escritorio basadas en Intel Pentium 4. Como era de esperar, el tiempo de ejecución fue proporcionalmente mayor en las PCs, puesto que son de menor capacidad de cómputo numérico. Sin embargo, también se tienen diferencias en el orden relativo de importancia de las funciones de acuerdo al tiempo de ejecución que cada una de ellas requiere. Si bien los cambios no son muy grandes, sí hay diferencias:

- La función de mayor tiempo de ejecución en las computadoras basadas en Pentium 4 es la que aparece como séptima en las computadoras basadas en Xeon. Por otro lado, la función de mayor tiempo de ejecución en las computadoras basadas en Xeon aparece como tercera en las computadoras basadas en Intel Pentium 4.
- Dado un orden en una de las plataformas de hardware (siempre según el tiempo de ejecución), el cambio de orden en la otra plataforma de hardware no supera una diferencia de 10 rutinas. Es decir que, si una rutina está en el *lugar* (orden de importancia según el tiempo de ejecución) i en una de las plataformas de hardware, esta misma rutina estará en el *lugar* $i \pm 10$ en la otra plataforma de hardware. Esta diferencia podría no considerarse significativa en el contexto de trabajo con más de 250 rutinas en total.

1.2. Redefiniendo Alcance y Objetivos

El volumen de código Fortran 77 con el cual se trabaja lleva casi inmediatamente a acotar el alcance de este trabajo en cuanto a la o las rutinas a analizar y modificar. Específicamente, se comenzará con la rutina de mayor tiempo de ejecución en las computadoras con múltiples núcleos, que tiene más de 850 líneas de código Fortran 77 a analizar/actualizar/optimizar etc.

Cada cambio que se propone en el código Fortran 77 (para *actualizarlo* a Fortran 90/95 o para mejorarlo u optimizarlo de alguna manera), será analizado desde el punto de vista numérico. Más específicamente, dada la complejidad del código a utilizar y los detalles específicos relativamente desconocidos de la implementación de la aplicación, los cambios realizados se aceptarán *solamente* si no cambian el resultado numérico. Este criterio exactitud numérica puede ser considerado como *muy* estricto, dado que al ser un modelo, los cambios no necesariamente significan un error ni mucho menos. Específicamente en el contexto de un modelo numérico del clima, estos cambios pueden resultar en una diezmilésima de diferencia en la temperatura ambiente, por ejemplo, y este cambio

- No necesariamente es un problema, dado que una variación tan pequeña no implica en sí misma que el resto del modelo proporcione cambios significativos.
- No necesariamente es un error, dado que el punto de comparación ya es una estimación numérica y, por lo tanto, en sí misma tiene errores. No es posible conocer *a priori* un criterio de evaluación en función del cual se determine la *exactitud* de una respuesta/resultado. Ni del modelo inicial programado en Fortran 77 ni del modelo que es el *resultado* de uno o más cambios sobre el modelo incial.

La caracterización/evaluación de los cambios numéricos que se puedan producir no es imposible. Sin embargo, se descarta a priori para avanzar sobre cambios que, evidentemente son absolutamente aceptables, dado que no producen ningún cambio numérico.

2. Estrategia General de Actualización de Código

Se planifican varias etapas de cambios, cada una de ellas con diferentes objetivos y niveles de dificultad. En todos los casos, los cambios se aplican a la rutina con mayores requerimientos de tiempo de cómputo y tienden a actualizar y optimizar el código fuente Fortran. Más específicamente, se producirá una versión *diferente* de la rutina (en un archivo .f) en cada etapa de cambios de manera tal que:

- Etapa 1: cambios sencillos, en parte de actualización sintáctica de Fortran 77 a Fortran 90/95 y en parte para mejorar la legibilidad de la codificación actual.
- Etapa 2: cambios directamente relacionados con el aprovechamiento de la expresividad de Fortran 90/95. Dado que todo el programa en general y la rutina a cambiar en particular determina diferentes tipos de operaciones entre vectores y matrices, la idea específica en este caso es aprovechar la notación de *arrays* de Fortran 90.
- Etapa 3: cambios directamente relacionados con la inclusión de directivas de OpenMP. Más que aprovechar las características de Fortran 90/95 la idea subyacente en este caso es optimizar el código para mejorar el rendimiento en las computadoras con procesadores de múltiples núcleos y/o SMP.

Como se aclara antes, todos los cambios se verificarán respecto de la salida numérica del programa sin ningún cambio de/en las rutinas *originales*.

Como parte de las tareas asociadas a los cambios, se intentará describir la complejidad de cada una de las etapas. La complejidad de las dos primeras etapas parecen particularmente interesantes, dado que tienen relación directa con la *actualización* de código originalmente escrito en Fortran 77 para producir código Fortran 90/95, con las mejoras que esta *nueva versión* de Fortran incorpora. Es importante recordar que aunque Fortran 90/95 incorpora muchas características útiles para el desarrollo de software, el trabajo estará *restringido* a la aplicación numérica original ya desarrollada/heredada. Puesto de otra forma, el aprovechamiento de las características de Fortran 90/95 depende del tipo de operaciones y/o procesamiento que se necesite hacer en la aplicación. En el caso del modelo climático con el que se trabaja, dificilmente se haga uso de las posibilidades de manejo de memoria dinámica, por ejemplo, dado que todo el modelo ya está programado haciendo uso de vectores y matrices cuyos espacios de memoria están asignado estáticamente y sin cambios en tiempo de ejecución.

3. Cambios de la Etapa 1: los Más Sencillos

Como se comenta antes, la primera etapa de cambios está orientada a actualizar el código Fortran 77 a Fortran 90/95 y, a la vez, mejorar la legibilidad de código. Dado que es el primer trabajo en detalle sobre el código existente de una rutina (la de mayor tiempo de ejecución), también en esta etapa se conocerán al menos algunos detalles específicos del tipo de operaciones y datos utilizados. Como en cierta forma *era esperable*, el procesamiento se da en una serie de bucles (*do loops*) mayormente con operaciones sobre matrices.

A partir de la apreciación previa de operaciones sobre datos organizados en matrices y utilizando la estructura de control de bucles, los cambios específicos que se proponen son:

- Todos los bucles deberían quedar expresados con la sintaxis `do ... end do`. Esto implicaría, al menos, cambiar los bucles de Fortran 77 con la sintaxis

  `do` *label var = expr1, expr2, expr3*
  statements
  label `continue`

  a la sintaxis `do ... end do` de Fortran 90/95.

- Todo el código debería tener alguna indentación. Actualmente solamente algunas secciones de código están indentadas, específicamente varios de los bloques de código (*cuerpo*) de los bucles.

- Solamente para mejorar la legibilidad, los bucles consecutivos se separarán con al menos una línea en blanco. Actualmente todo el código es consecutivo (sin líneas en blanco), incluyendo secuencias de líneas de comentarios y bucles que no necesariamente están relacionados.

- Reemplazar cada aparición de `goto` por código equivalente y estructurado (utilizando control de flujo `if`, por ejemplo).

Durante los cambios que se realizaban para que todos los bucles estén expresados con la sintaxis `do ... end do` de Fortran 90/95, se encontraron también secciones de código (bucles) con "shared do loop termination", que es considerada una característica obsoleta en [1] [7].

3.1. Ejemplos de Cambios Específicos

La mayor parte de los cambios en la rutina de mayor tiempo de ejecución fueron muy sencillos, aunque fueron necesarios a lo largo de las más de 800 líneas de código. Estos cambios sencillos corresponden a secuencias de código como el de la Fig. 1, que se cambió a (utilizando `do ... end do` e indentando el

```
      do 345 i=1,imax
      ctmp(i,1)=one
      ctmp2(i,1)=1.
      ctmp3(i,1)=1.
345   continue
```

Figura 1: Bucle con *Label* y sin Indentación

cuerpo del bucle) tal como se muestra en la Fig. 2.

```
do i=1,imax
  ctmp(i,1)=one
  ctmp2(i,1)=1.
  ctmp3(i,1)=1.
end do
```

Figura 2: Bucle sin *Label* y con Indentación

Aunque no era la mayoría del código, también había partes de la rutina con bucles anidados como el de la Fig. 3, donde se puede *ganar* mucho en legibilidad siguiendo las pautas mencionadas antes, y *produciendo* el código equivalente (es decir con la misma semántica y, de hecho, con los mismos resultados de procesamiento numérico verificado en tiempo de ejecución), tal como se muestra en la Fig. 4. Aunque mucho menos frecuente que los casos anteriores, también había en el código original casos como el que se

```
      do 319 k=2,l
      do 317 i=1,imax
      topm(i,k)=topm(i,k-1)+phitmp(i,k)
      topphi(i,k)=topphi(i,k-1)+psitmp(i,k)
317   continue
319   continue
```

Figura 3: Bucles Anidados con *Labels* y sin Indentación

```
do k=2,l
  do i=1,imax
    topm(i,k)=topm(i,k-1)+phitmp(i,k)
    topphi(i,k)=topphi(i,k-1)+psitmp(i,k)
  end do
end do
```

Figura 4: Bucles Anidados sin *Labels* y con Indentación

```
      do 101 k=1,l
      do 101 i=1,imax
      x(i,k)=temp(i,k)-h25e2
      y(i,k)=x(i,k)*x(i,k)
101   continue
```

Figura 5: Bucles Anidados con *Labels* y con "Shared do Loop Termination"

muestra en la Fig. 5, es decir con "shared do loop termination", que se *transformó* en el código mostrado en la Fig. 6 y tampoco en este caso se alteran los resultados numéricos como, además, era *esperable*. Es

```
do k=1,l
  do i=1,imax
    x(i,k)=temp(i,k)-h25e2
    y(i,k)=x(i,k)*x(i,k)
  end do
end do
```

Figura 6: *Eliminación* de "Shared do Loop Termination"

interesante que todos los cambios produjeron código binario (o *código objeto*) diferente, aunque algunos de estos cambios realmente no parecen significativos como para que el compilador genere diferente código binario. Por otro lado, todos estos cambios son *estrictamente* sintácticos y, además, se estima que pueden ser realizados de manera automática. Si bien no se aplicó/implementó esta idea de cambios automáticos, no parece ser muy complejo hacerlo, dado que básicamente se deben identificar patrones sencillos de codificación y reemplazarlos por otros que también son sencillos.

Aunque en la rutina de mayor tiempo de ejecución no se utiliza `goto`, sí aparece en varias otras rutinas. De hecho, en la rutina de mayor tiempo de ejecución en las computadoras basadas en Intel Pentium 4 sí se utiliza `goto` y además es mucho más frecuente la utilización de "shared do loop termination". Solamente para experimentar, se analizaron los cambios necesarios para reemplazar `goto` por `if` y es bastante sencillo, aunque se estima que no sería sencillo hacerlo de manera automática.

Quizás como *efectos colaterales* de la aplicación de todos estos cambios tan sencillos, no solamente se mejora la legibilidad de la rutina sino que

- Se tuvo el primer contacto con el código fuente de la rutina, conociendo algunos detalles interesantes en cuanto a la sucesión de bucles que implica el procesamiento.
- Se puede estimar que la mayoría de (sino todos) estos cambios se podrían haber realizado con algún tipo de análisis sintáctico (*parsing*) sobre el código de manera automática.

Entre las características *interesantes* del código específico de la rutina, se pueden mencionar:

- En el principio se incluyen (utilizando `include`), 11 archivos .f, la gran mayoría de ellos declarando varias áreas `common` (sin código ejecutable).
- Aunque no sucede en la mayoría de los casos, se hace uso de la declaración implícita de datos de Fortran 77. Se eligió no realizar la declaración explícita de todos los datos por la cantidad de cambios necesarios no solamente en esta rutina sino también en todos los .f que se incluyen.
- En algunos casos, se cambia la forma de acceso a los datos, específicamente se acceden *arrays* bidimensionales (matrices) como si fueran *unidimensionales* (vectores).

Un ejemplo de esto último se puede ver en que, con las declaraciones que se muestran en la Fig. 7 luego

```
      common / vtemp / phitmp(imax,l),psitmp(imax,l),tt(imax,l),
     &          fac1(imax,l),fac2(imax,l),
     &          ctmp(imax,lp1),x(imax,l),y(imax,l),
     &          topm(imax,l),topphi(imax,l),
     &          ctmp3(imax,lp1),ctmp2(imax,lp1)
      common /vtemp/ dummy(imax*(2*l*l+5*l+16))
      double precision tt,fac1
      dimension f(imax,l),ff(imax,l),ag(imax,l),agg(imax,l)
```

Figura 7: Bloques Common y Declaraciones

se tiene el bucle de la Fig. 8

```
      do 301 i=1,imax*l
      f(i,1)=h44194m2*(apcm(1)*x(i,1)+bpcm(1)*y(i,1))
      ff(i,1)=h44194m2*(atpcm(1)*x(i,1)+btpcm(1)*y(i,1))
      ag(i,1)=(h1p41819+f(i,1))*f(i,1)+one
      agg(i,1)=(h1p41819+ff(i,1))*ff(i,1)+one
      phitmp(i,1)=var1(i,1)*(((( ag(i,1)*ag(i,1))**2)**2)**2)
      psitmp(i,1)=var2(i,1)*(((( agg(i,1)*agg(i,1))**2)**2)**2)
301   continue
```

Figura 8: Acceso a Matrices como Arreglos de una Columna

Es decir que inicialmente, en la declaración, las variables `f`, `ff`, `ag`, `agg`, `phitmp` y `psitmp` son declaradas como *arrays* de dos dimensiones, con `imax` filas y `l` columnas. Pero en el bucle que se muestra, todos los elementos se acceden *como si estuvieran* en la primera columna y *como si esta primera columna tuviera* `imax*l` elementos.

4. Cambios de la Etapa 2: *Hacia* Fortran 90/95

Como se comenta antes, la segunda etapa de cambios está orientada a aprovechar específicamente la expresividad de Fortran 90/95 referente a la notación de arrays [4]. En este contexto, se puede aprovechar el conocimiento adquirido en la etapa anterior, dado los bucles se utilizan para, justamente, operar sobre los arrays sl de la rutina. En esta etapa, los cambios específicos que se proponen son:

- Utilizar notación de arrays de manera extensiva, es decir en todas las secuencias de código donde sea posible.
- Mejorar algunos detalles *menores* del código, específicamente relacionados con las expresiones dentro de los bucles. Esto es posible dado que la utilización de notación de arrays implica interiorizarse de los detalles del procesamiento dentro de los bucles (para identificar las operaciones utilizadas y los elementos de los arrays a los que se hace referencia).

En general, todos los bucles son bastante sencillos para analizar en cuanto al procesamiento que se realiza en cada uno de ellos. También es es sencillo el análisis de las expresiones dentro de los bucles para determinar si es posible la utilización de notación de arrays con el objetivo de aprovechar la expresividad de Fortran 90/95 y simplificar el código fuente. Entre los cambios más sencillos a realizar incorporando

```
do i=1,imax
  ctmp(i,1)=one
  ctmp2(i,1)=1.
  ctmp3(i,1)=1.
end do
```

Figura 9: Inicialización en una Columna

notación de arrays están los que corresponden a código como se muestra en la Fig. 9 (nótese que ahora todos los bucles tienen la sintaxis `do` ... `end do`), donde `one` no es más que una constante, y todo el bucle se puede expresar como lo muestra la Fig. 10.

```
ctmp(1:imax, 1)  = one
ctmp2(1:imax, 1) = 1.
ctmp3(1:imax, 1) = 1.
```

Figura 10: Inicialización en una Columna con Notación de *Arrays*

Y dado que las declaraciones de los arrays involucrados son (en cuanto a los índices) las que se muestran en la Fig. 11, se pueden utilizar directamente las asignaciones que se muestran en la Fig. 12. Entre los

```
ctmp(imax,lp1)
...
ctmp3(imax,lp1),ctmp2(imax,lp1)
```

Figura 11: Declaraciones de *Arrays*

```
ctmp(:, 1)  = one
ctmp2(:, 1) = 1.
ctmp3(:, 1) = 1.
```

Figura 12: Inicialización de una Columna con Notación de *Arrays*

ejemplos de bucles más comunes para la utilización de notación de arrays se muestra en la Fig. 13, que

```
do i=1,imax
  topm(i,1)=phitmp(i,1)
  topphi(i,1)=psitmp(i,1)
end do
```

Figura 13: Asignación de una Columna en un Bucle

se *transforma* en el código de la Fig. 14, dado que todas las variables (matrices) involucradas tienen los mismos índices de fila.

```
topm(:,1)   = phitmp(:,1)
topphi(:,1) = psitmp(:,1)
```

Figura 14: Asignación de una Columna con Notación de *Arrays*

Todos estos ejemplos corresponden a lo que se podría denominar la parte de inicialización de la subrutina. En general, pueden aparecer secuencias de procesamiento (dentro de bucles) un poco más complejas. En particular, uno de los ejemplos ya mencionados en la Fig. 8 se muestra en la Fig. 15 ahora ya *transformado* con sintaxis `do` ... `end do` y tiene esta característica de mayor complejidad en cuanto a las expresiones involucradas en la utilización de notación de *arrays*. En este caso, además, se accede matrices de `imax x l` elementos (`imax` filas y `l` columnas) *como si fueran* de `imax*l x 1` elementos (`imax*l` filas y `1` columna).

```
do i=1,imax*l
  f(i,1)=h44194m2*(apcm(1)*x(i,1)+bpcm(1)*y(i,1))
  ff(i,1)=h44194m2*(atpcm(1)*x(i,1)+btpcm(1)*y(i,1))
  ag(i,1)=(h1p41819+f(i,1))*f(i,1)+one
  agg(i,1)=(h1p41819+ff(i,1))*ff(i,1)+one
  phitmp(i,1)=var1(i,1)*(((( ag(i,1)*ag(i,1))**2)**2)**2)
  psitmp(i,1)=var2(i,1)*(((( agg(i,1)*agg(i,1))**2)**2)**2)
end do
```

Figura 15: Procesamiento sobre Matrices como Arreglos de una Columna

La Fig. 16 no solamente muestra el resultado de utilizar notación de arrays sino también la simplificación de la expresión que aparece en la Fig. 15

`(((( ag(i,1)*ag(i,1))**2)**2)**2)`

utilizando directamente exponenciación: `**16`. En principio, se estimaba que el resultado numérico podría cambiar dado este último cambio (de una multiplicación y varias potencias de 2 sucesivas a una única potencia de 16), pero no fue así, el resultado numérico es el mismo. Quedaría pendiente la posibilidad

```
 f(1:imax*l,1)=h44194m2*(apcm(1)*x(1:imax*l,1)+
&                bpcm(1)*y(1:imax*l,1))
 ff(1:imax*l,1)=h44194m2*(atpcm(1)*x(1:imax*l,1)+
&                 btpcm(1)*y(1:imax*l,1))
 ag(1:imax*l,1)=(h1p41819+f(1:imax*l,1))*f(1:imax*l,1)+one
 agg(1:imax*l,1)=(h1p41819+ff(1:imax*l,1))*ff(1:imax*l,1)+one
 phitmp(1:imax*l,1)=var1(1:imax*l,1)*(ag(1:imax*l,1)**16)
 psitmp(1:imax*l,1)=var2(1:imax*l,1)*(agg(1:imax*l,1)**16)
```

Figura 16: Notación de *Arrays* sobre Matrices como Arreglos de una Columna

de utilizar los *arrays* sin hacer mención explícita a conjuntos de filas y/o columnas (de manera similar a lo que se muestra en la Fig. 12 y en la Fig. 14), pero esto llevaría a un análisis más detallado de la declaración de todas las variables involucradas en todas las expresiones. Se descarta esta tarea para continuar con otras posibilidades que, en principio, son más sencillas de resolver.

La Fig. 17 muestra un caso de expresiones con *arrays* de dos dimensiones en dos bucles anidados. Aunque los límites de las variables de control de los bucles (`i` y `k`) implican que no se asignan todos los elementos de los *arrays* `topm` y `topphi`, la codificación del procesamiento en este segmento de código es relativamente sencilla y se muestra en la Fig. 18. Como era *esperable*, la simplificación es notable a simple

```
do k=2,l
  do i=1,imax
    topm(i,k)=topm(i,k-1)+phitmp(i,k)
    topphi(i,k)=topphi(i,k-1)+psitmp(i,k)
  end do
end do
```

Figura 17: Procesamiento en Dos Bucles Anidados

vista. También en este caso se comprobó (con experimentación) que el resultado numérico no cambia. A partir de estos ejemplos también se estima que la gran mayoría de los cambios *parecen ser automáticos*, es decir que pueden realizarse analizando los límites de las varibles de control de los bucles y los índices

```
topm(:, 2:l)   = topm(:, 1:l-1) + phitmp(:, 2:l)
topphi(:, 2:l) = topphi(:, 1:l-1) + psitmp(:, 2:l)
```

Figura 18: Procesamiento de Dos Bucles Anidados con Notación de *Arrays*

(de los *arrays*) de las expresiones involucradas en el cuerpo de los mismos. Más específicamente:

1. Identificar los límites de las variables de control de los bucles.
2. Identificar el uso de las variables de control de los bucles como índices de los *arrays*.
3. Identificar el primer y último valor de índice de cada una de las dimensiones de los *arrays* utlizados.
4. Indicar en cada una de las dimensiones el primer y último valor de cada índice.

En el caso específico del modelo numérico utilizado, cada una de estas tareas es relativamente sencilla dado que el procesamiento se da en una secuencia de bucles, cada bucle tiene a lo sumo diez (aproximadamente) instrucciones, casi no hay selecciones condicionales (tales como `if`) y en ningún caso hay más de dos bucles anidados. Es claro que todas estas características simplifican notablemente el análisis y la *recodificación* y no se pueden asegurar estas características en general, para cualquier programa en Fortran 77.

Un último ejemplo que valdría comentar es el del código que se muestra en la Fig. 19, que tiene dos bucles anidados y en las expresiones se incluyen *arrays* de dos dimensiones y variables escalares (`h25e2`). En este caso, también es sencillo seguir la serie de pasos dados antes para analizar el contenido de los

```
do k=1,l
  do i=1,imax
    x(i,k)=temp(i,k)-h25e2
    y(i,k)=x(i,k)*x(i,k)
  end do
end do
```

Figura 19: Dos Bucles Anidados con *Arrays* y Variables Escalares

bucles y expresar el procesamiento con notación de arrays, tal como lo muestra la Fig. 20. En este caso

```
x(1:imax, 1:l) = temp(1:imax, 1:l) - h25e2
y(1:imax, 1:l) = x(1:imax, 1:l)**2
```

Figura 20: Notación de *Arrays* Incluyendo Variables Escalares

específico, las declaraciones de las variables involucradas en las asignaciones/procesamiento `x`, `y` y `temp` son tales que

- `x` es un *array* de `imax` filas y `l` columnas.
- `y` es un *array* de `imax` filas y `l` columnas.
- `temp` está declarado en uno de los .f que se inlcuyen al principio de la rutina y es un *array* de `imax` filas y `lp1` columnas.

Es decir que sería *inmediato* expresar la línea

```
y(1:imax, 1:l) = x(1:imax, 1:l)**2
```

de la Fig. 19 *directamente* como

```
y = x**2
```

Pero habría que analizar con más detalle la relación entre `l` y `lp1` (específicamente si tienen el mismo valor) para determinar si es posible una *simplificación* similar en la línea

```
x(1:imax, 1:l) = temp(1:imax, 1:l) - h25e2
```

Este análisis no se llevó a cabo y se deja pendiente para cuando se realice un cambio más profundo, con el análisis de todos los .f incluidos para esta rutina y las definiciones de cada uno de ellos. Finalmente, el código de la Fig. 20 quedó *simplificado* con notación de *arrays* como se muestra en la Fig. 21.

Es importante recordar que la notación de *arrays* no solamente tiene mayor poder expresivo que la notación con valores escalares (la *estándar*, accediendo a los valores individuales de un *array*, por ejemplo) sino que también reduce la longitud del código Fortran. Considerando que hay más de 100 bucles en la rutina sobre la que se trabajó, y que cada bucle tiene usualmente las dos líneas de código *necesarias*

```
x(:, 1:l) = temp(:, 1:l) - h25e2
y = x**2
```

Figura 21: Simplificación de Indices Utilizando Notación de *Arrays*

para `do` y `end do`, la rutina se reduciría *automáticamente* en 200 líneas como *efecto colateral* de la utilización de la notación de *arrays*. Esto significaría una reducción de casi 30 % en la *longitud* de la rutina medida en líneas de código Fortran. Esta reducción en cantidad de líneas no necesariamente en buena en sí misma, pero demuestra que el código resultante es, al menos, más compacto/reducido manteniendo la misma funcionalidad. De hecho, los experimentos mostraron que mantienen exactamente la misma salida numérica. Sin embargo, dada la cantidad de bucles de la rutina, solamente algunos de ellos fueron expresados con notación de *arrays*, dejando el resto para un siguiente cambio más exhaustivo/completo. Por otro lado, todos los bucles son similares a los que se han mostrado hasta este punto y se podrían expresar con notación de *arrays* de maneras similares a las explicadas, es decir que no se han encontrado ejemplos más complejos al menos en la rutina sobre la que se trabajó.

5. Cambios de la Etapa 3: *Incorporación* de OpenMP

Como se comenta sobre el principio de este reporte técnico, la tercera etapa de cambios está enfocada a la inclusión de directivas de OpenMP. Esta etapa específica es la que tiene como objetivo mejorar el rendimiento en las computadoras con múltiples núcleos y/o SMP, reduciendo el tiempo de ejecución en principio de una rutina y, como consecuencia, reduciendo el tiempo de ejecución total de manera proporcional. En este punto se intentarán aprovechar no solamente las posibilidades de ejecución en múltiples CPUs sino también el conocimiento adquirido en las dos etapas de cambios anteriores respecto del código Fortran con el que se trabaja.

Específicamente con respecto a la utilización de OpenMP, la estrategia general es, en principio, incorporar paralelismo en la rutina utilizando las directivas OpenMP

1. `WORKSHARE`, en todas las operaciones expresadas con notación de *arrays* [3]. En este sentido, se aprovecharían *directamente* los cambios de la etapa anterior, que justamente incorporaron notación de *arrays* reemplazando procesamiento hecho en los bucles.

2. `DO`, en todos los bucles que quedaron en la rutina, dado que no todos se expresaron con notación de *arrays*, tal como se aclara en el final de la sección anterior.

Los ejemplos específicos de utilización de la directiva `WORKSHARE` serían varios de los que se muestran en la sección anterior. En el código de la Fig. 12, por ejemplo, se podría incorporar la directiva `WORKSHARE` tal como se muestra esquemáticamente en la Fig. 22. Se debe recordar que la directiva `WORKSHARE` debe

```
!$OMP PARALLEL
...
!$OMP WORKSHARE
      ctmp(:, 1)  = one
      ctmp2(:, 1) = 1.
      ctmp3(:, 1) = 1.
!$OMP END WORKSHARE
...
!$OMP END PARALLEL
```

Figura 22: Inicialización de una Columna con WORKSHARE

estar *dentro del alcance* de la directiva `PARALLEL`, tal como se muestra, justamente, en la Fig. 22. En el caso de no tener la directiva `!$OMP PARALLEL` precediendo a la `!$OMP WORKSHARE`, se podría utilizar *directamente* `!$OMP PARALLEL WORKSHARE`, tal como se muestra en la Fig. 23. Más específicamente, el código de la Fig. 23 es equivalente al de la Fig. 22 desde el punto de vista de la cantidad de *threads* que ejecutan el código con notación de *arrays* dentro del *alcance* de la directiva `WORKSHARE`. Sin embargo, no es equivalente en cuanto a la creación y terminación de *threads*, lo cual puede generar diferencias de rendimiento. La directiva `WORKSHARE` provee, en cierto sentido, paralelización *automática* de la mayoría

```
!$OMP PARALLEL WORKSHARE
      ctmp(:, 1)  = one
      ctmp2(:, 1) = 1.
      ctmp3(:, 1) = 1.
!$OMP END PARALLEL WORKSHARE
```

Figura 23: Inicialización de una Columna con PARALLEL WORKSHARE

de los ejemplos de utilización de la notación de *arrays* dados en la sección anterior. Sin embargo, no siempre que se tiene una expresión con notación de *arrays* es posible utilizar la directiva `WORKSHARE`. Un ejemplo específico es el de la Fig. 18, donde los valores de algunos elementos de las matrices involucradas dependen de otros de las mismas matrices y, por lo tanto, no se podría ejecutar este código en diferentes *threads*.

Solamente como otro ejemplo de la utilización de la directiva `WORKSHARE`, se muestra el código de la Fig. 21 incorporando esta directiva, en la Fig. 24. Es interesante notar que cuando no se hace referencia

```
!$OMP PARALLEL WORKSHARE
      x(:, 1:l) = temp(:, 1:l) - h25e2
      y = x**2
!$OMP END PARALLEL WORKSHARE
```

Figura 24: Paralelización de Operaciones con Matrices con PARALLEL WORKSHARE

a los índices en la notación de *arrays*, el código Fortran 90/95 sí es automáticamente paralelizable con OpenMP. Esto se debe a que no podría haber dependencias de datos entre diferentes valores de diferentes índices, dado que a cada índice de de la parte izquierda de la asignación le *corresponde el mismo* valor de índice del *array* (o de los *arrays*) de la parte derecha de la asignación, o valores escalares. En cierto modo, esto se puede aprovechar para *automatizar* el proceso de paralelización con OpenMP, aunque no necesariamente siempre es posible expresar las operaciones con notación de *arrays* sin hacer referencia a los índices.

Aunque sintácticamente es posible incluir la directiva `WORKSHARE`, el compilador de Fortran utilizado (de Intel) no la tiene en cuenta para paralelizar y/o distribuir el cómputo con *arrays* entre varios *threads* de ejecución [6]. Sin embargo, como se aclara en el principio de esta sección, se analizará la paralelización de los bucles con la directiva OpenMP `DO`, dado que no todos los bucles se expresaron con notación de arrays en la etapa de cambios anterior. Si todos los bucles hubieran sido expresados con notación de *arrays*, sería necesario paralelizar la versión anterior a la segunda etapa de cambios o directamente cambiar el compilador de Fortran (tomando el trabajo necesario para el cambio del manejo numérico de los archivos intermedios que se generan y utilizan). En cualquier caso, el análisis de utilización de la directiva `DO` que sigue es independiente del uso de `WORKSHARE`, dado que está enfocado en la posible paralelización del procesamiento. Con este análisis se intenta determinar básicamente de la dependencia de datos de los cálculos y no la utilización de una u otra directiva OpenMP, al menos en el caso específico de la rutina que se analiza.

Desde la perspectiva de utilización de OpenMP, la rutina tiene una serie de bucles, algunos son posibles de paralelizar y otros no. La mayoría de los bucles son paralelizables, y se podría aprovechar un *patrón* en esta serie de bucles, donde cada aproximadamente 7 bucles hay uno que no es paralelizable, y esta serie se repite. Es interesante que no hay procesamiento fuera de los bucles, es decir que, en principio, podría pensarse en utilizar OpenMP como lo muestra esquemáticamente la Fig. 25, donde se incluye una directiva `!$OMP PARALLEL DO` *para* los bucles paralelizables y se deja el resto de los bucles sin ninguna directiva OpenMP.

Otra alternativa, o en realidad una mejora sobre esta primera propuesta, consiste en identificar las posibles series de bucles *sucesivos* que son paralelizables e inclirlos dentro de una región paralela con `OMP PARALLEL` y, para cada uno de los bucles individuales, poner una directiva `DO`. Esta alternativa se muestra en la Fig. 26, donde no solamente los bucles con directiva `DO` tienen que ser *paralelizables* sino que entre dos de estos bucles no podría haber uno no paralelizable. El o los bucles no paralelizables deben estar fuera de las secciones paralelas (fuera del alcance estático o léxico según [5]) `OMP PARALLEL` y `OMP END PARALLEL`. La alternativa de la Fig. 26 puede considerarse una mejora desde el punto de vista de rendimiento sobre la de la Fig. 25 dado que los *threads* de ejecución paralela se crean cuando el control llega a la directiva `OMP PARALLEL` y a partir de allí *solamente* se distribuye el trabajo de los bucles entre

```
!$OMP PARALLEL DO
      do
        ...
      end do
!$OMP PARALLEL DO
      do
        ...
      end do
...
      do
        ...
      end do
...
```

Figura 25: Esquema General de Paralelización con OpenMP PARALLEL DO

```
!$OMP PARALLEL
!$OMP DO
      do
        ...
      end do
!$OMP DO
      do
        ...
      end do
!$OMP END PARALLEL
...
      do
        ...
      end do
...
```

Figura 26: Esquema General de Paralelización con OpenMP DO Sucesivos

los diferentes *threads* con cada directiva `DO`. En el caso de la Fig. 25, es decir utilizando la directiva `OMP PARALLEL DO`, se crean y terminan *threads* de ejecución para cada uno de los bucles donde se incluye esta directiva.

Hasta este punto, se podrían ver las propuestas presentadas como dos formas diferentes de *secuencializar* la ejecución de los bucles no paralelizables, que son la *minoría* de los bucles dentro de la serie de bucles de la rutina. Desde este punto de vista, habría al menos otras dos maneras de *secuencializar* la ejecución de los bucles que no son paralelizables:

- Con la directiva `MASTER`, que indica que solamente el *master thread* debe ejecutar un bloque de código.
- Con la directiva `SINGLE`, que indica que solamente un *thread* debe ejecutar un bloque de código.

Se puede afirmar en este caso que es preferible la directiva `SINGLE`, dado que implica una sincronización *barrier* al final de la misma, es decir que todos los *threads* continúan a partir del `END SINGLE` solamente cuando el único *thread* que ejecuta el bloque termina. La Fig. 27 muestra la forma de secuencializar los bucles no paralelizables con la directiva `SINGLE`. Es interesante notar que no es importante cuál *thread* ejecuta los bucles no paralelizables, dado que las variables involucradas son compartidas (*shared*) por todos los *threads* y, por lo tanto, todos los cambios son visibles por todos los *threads*. Por otro lado, sí es importante que todos los *threads* ejecuten sincronizadamente a continuación de los bucles que no son paralelizables por las dependencias de datos, y por lo tanto se considera que la directiva `SINGLE` es la más apropiada por la sincronización implícita del `END SINGLE`. Por otro lado, no hay posibilidad de que el *thread* que ejecuta esta parte del código afecte a los demás, dado que todos llegan a esta parte del código habiendo finalizado todos los bucles anteriores por la sincronización implícita de la directiva `DO` (todos los *threads* se sincronizan al final). Desde la perspectiva de rendimiento, se podría afirmar que esta propuesta

```
!$OMP PARALLEL
! Bucle paralelizable
!$OMP DO
      do
        ...
      end do
...
! Bucle no paralelizable
!$OMP SINGLE
      do
        ...
      end do
!$OMP END SINGLE
...
!$OMP END PARALLEL
```

Figura 27: Esquema General de Paralelización con OpenMP DO Sucesivos y SINGLE

es de las *mejores*, dado que se crean los *threads* al principio de la ejecución de la rutina y se terminan al final. Por lo tanto, no habría creación y finalización de *threads* en el código/bucles *intermedio/s* de la rutina, con su consiguiente *sobrecarga*.

Hasta este punto se presentaron varias alternativas de paralelización con OpenMP, se analizaron, y se seleccionó la que sería de las mejores desde el punto de vista del rendimiento porque:

- Paraleliza todos los bucles posibles. Esto significaría que no habría una mejor alternatica en cuanto a distribución de procesamiento entre threads de ejecución simultánea en múltiples núcleos o SMP. Esta afirmación se sustenta en el análisis detallado del procesamiento de la rutina.
- Minimiza la creación y finalización de *threads*, con una directiva `OMP PARALLEL` al principio y una directiva `OMP END PARALLEL` al final del procesamiento de la rutina. Se presentó una propuesta que *secuencializa sincronizadamente* los bucles que no son paralelizables, de forma tal que todos los bucles no paralelizables se ejecuten en el orden correcto para no afectar el procesamiento de la rutina desde el punto de vista numérico.

Quedaría por analizar experimentalmente no solamente la validez numérica de los resultados sino también el rendimiento obtenido en computadoras con múltiples núcleos o SMP. Es interesante resaltar en este punto que, *a priori*, no debería haber ningún cambio numérico respecto de la rutina original, no paralelizada con OpenMP. Esto se debe a que no solamente no se paralelizan los bucles que producirían resultados diferentes de los *originales* (los que tienen dependencias *internas* de datos, por ejemplo) sino que ni siquiera se cambia el orden de las operaciones entre escalares. Más específicamente: no se utilizan, por ejemplo, reducciones, que pueden producir resultados diferentes de acuerdo al orden en que se opera sobre una serie de números (por efectos de redondeo, por ejemplo).

En lo que respecta al rendimiento esperado, es importante recordar que la mejora puede ser considerada poco significativa respecto del tiempo total de procesamiento. En cierta forma, esto puede ser desalentador dada la longitud de todo el análisis y trabajo reflejado en este reporte técnico, pero no se puede sacar de contexto el trabajo realizado para incluir directivas de OpenMP a una rutina en particular. Como se aclara en la introducción, la rutina que tiene mayor tiempo de ejecución se ejecuta durante el 13 % del tiempo total. Es decir que se está reduciendo el tiempo de ejecución en *el contexto* del 13 % del total. Si, por ejemplo, se lograra distribuir el cómputo de esta rutina en dos CPUs y no hubiera ninguna *penalización* por el tiempo necesario para la gestión de los *threads* (creación, sincronización y terminación, por ejemplo), esta rutina pasaría del 13 % del tiempo total al 6.5 % del tiempo total. Si, por ejemplo, el tiempo total *original* era de 100 segundos, el tiempo total pasaría a ser de 93.5 segundos, es decir que se mejoraría un 6.5 % a pesar de tener disponibles 2 CPUs. Sin embargo, esta serie de cambios podría aplicarse, eventualmente, a todo el programa (o al menos a las rutinas con mayor tiempo de ejecución) y en ese caso sí se podría reducir el tiempo total de manera proporcional a la cantidad de CPUs utilizadas. Esta *escala global* de cambios está fuera del alcance de este reporte técnico.

5.1. Resultados de la Experimentación

La primera evaluación que se llevó a cabo con la experimentación fue estrictamente numérica. En este sentido, esta rutina en particular no presenta inconvenientes justamente por lo explicado antes: la paralelización no genera resultados numéricamente diferentes, dado que no hay operaciones que puedan ser afectadas en este sentido. En realidad, la evaluación de resultados numéricos fue utilizada para la verificación de que el análisis había sido hecho correctamente y, además, que este análisis se implementa correctamente con las directivas OpenMP. La evaluación de resultados numéricos es sencilla: se comparan todas las salidas generadas sin OpenMP con todas las que se generan con OpenMP. Deben ser idénticas en todos los sentidos: la misma cantidad de resultados numéricos y los mismos valores numéricos de los resultados. El programa con el que se trabaja genera aproximadamente 170 archivos, con un total de aproximadamente 14 MB. Por lo tanto, se considera que se tiene igual salida cuando hay igual cantidad de archivos con cada archivo de igual longitud y con igual contenido. Dado que lo importante es conocer si toda la salida es *exactamente* igual, se puede utilizar (y de hecho se utilizó) directamente el comando `diff`. Como era *esperable* en caso de haber implementado correctamente el análisis anterior para la utilización de las directivas OpenMP, los resultados fueron idénticos al utilizar OpenMP con más de un *thread* (`OMP_NUM_THREADS` mayor que uno).

Una vez verificados satisfactoriamente los resultados numéricos, es interesante el análisis de rendimiento obtenido a partir de la utilización de OpenMP. En realidad, todo lo que habría que hacer consiste en analizar los tiempos de ejecución a medida que se utiliza mayor cantidad de *threads* e identificar si la sobrecarga de utilizar OpenMP (con el tiempo necesario para la gestión de los *threads*: creación, sincronización y terminación, por ejemplo), no *enmascara* o es comparable con el tiempo de ejecución que se reduce al paralelizar los cálculos de la rutina sobre la que se trabajó. La Tabla 1 muestra los resultados obtenidos para diferentes valores de `OMP_NUM_THREADS` (*threads* de ejecución). La primera columna

CPUs	% Tiempo	Núm. de Orden
1	13	1
2	10.74	1
4	5.88	3
8	3.85	7

Tabla 1: Resultados

de esta tabla indica la cantidad de CPUs utilizadas (específicamente asignando la variable de entorno `OMP_NUM_THREADS`), la segunda columna muestra el porcentaje de tiempo del total de la rutina sobre la que se trabajó, y la tercera columna muestra el orden relativo de esta rutina respecto de las demás teniendo en cuenta el porcentaje total de tiempo utilizado. A partir de los resultados que se muestran en la Tabla 1, se puede identificar no solamente que el tiempo relativo es menor a medida que se utilizan más CPUs sino que también comienza a ser relevante la utilización de OpenMP en otras rutinas. Cuando se utilizan 8 CPUs, por ejemplo, se tienen otras 6 rutinas que utilizan más tiempo de ejecución que la rutina sobre la que se ha trabajado. Quizás se requiere un análisis más detallado para interpretar los resultados de rendimiento directamente relacionados con la cantidad de CPUs utilizadas. Más específicamente, por qué, por ejemplo, cuando se utilizan 2 CPUs se reduce solamente al 10.74% del tiempo en vez de a valores cercanos al 6.5 %. Sin embargo, en porcentajes tan pequeños del total quizás no sería importante un análisis tan detallado de resultados. Sin embargo, este análisis detallado sería muy necesario cuando la cantidad de tiempo relativo (y/o la cantidad de rutinas *afectadas*) sea mayor.

6. Conclusiones y Trabajo Futuro

El trabajo con código heredado (*legacy code*) se suele caracterizar como muy complejo, aunque por supuesto esta *complejidad* depende de muchos factores. En el caso específico de Fortran es toda un área de trabajo, dado que, entre otras cosas:

- Es uno de los pocos lenguajes con una larga historia de software en producción, con múltiples aplicaciones en producción desde hace varios años, y en algunos casos, décadas.
- Tiene múltiples aplicaciones desarrolladas durante varios años, por diferentes profesionales de diferentes áreas de aplicación.

- Muchos de los programas (sino *todos*) en ambientes de producción son aplicaciones numéricas, es decir que *provienen* de un modelo matemático de un proceso físico/real que se resuelven por métodos numéricos en un programa.

El caso del modelo climático con el que se trabajó para este reporte técnico es quizás representativo de todas estas características. Solamente como ejemplo: varias de las subrutinas fueron programadas hace casi 15 años en Fortran 77.

Con respecto a los cambios o a la actualización de código Fortran 77 a Fortran 90/95, se puede reconocer que es detallado, pero no es necesariamente complicado en sí mismo. Como siempre, depende del tipo de programación que se haya utilizado (múltiple utilización de `goto` o no, por ejemplo). A priori, se podría estimar que la mayoría de las rutinas a optimizar es similar a la que se comenta en este reporte técnico, pero esto no es más que una estimación. Específicamente relacionado con la notación de *arrays*, es importante señalar que el código fuente gana en expresividad y además se puede reducir notablemente en líneas de código, lo cual suele mejorar la legibilidad de código.

La incorporación de directivas OpenMP en la rutina con la que se trabajó fue particularmente sencilla desde el punto de vista conceptual. Sin embargo, también el trabajo para estos cambios fue muy detallado, individualizando todos los bucles uno a uno en cuanto a si son paralelizables o no y, luego, la *secuencialización* de la ejecución de estos bucles no paralelizables.

Se podría afirmar que la actualización de código Fortran 77 (*legacy code*) a Fortran 90/95 quizás sea muy minuciosa, pero provee experiencia importante para la incorporación de directivas OpenMP. También es importante resaltar que, a menos que el tiempo de ejecución esté *concentrado* en muy pocas rutinas y/o código fuente, la utilización de OpenMP para una reducción importante de tiempo de ejecución será también una tarea minuciosa. Sin embargo, el trabajo de análisis para la incorporación de OpenMP parece *simplificado* a la identificación de los bucles que son paralelizables y los que no lo son.

Referencias

[1] American National Standards Institute, *ANSI X3.198-1992 (R1997)*. American National Standard - Programming Language Fortran Extended, informally known as Fortran 90.

[2] R. Chandra, R. Menon, L. Dagum, D. Kohr, D. Maydan, J. McDonald, Parallel Programming in OpenMP, Morgan Kaufmann, 2000, ISBN 1558606718.

[3] B. Chapman, G. Jost, R. van der Pas, Using OpenMP: Portable Shared Memory Parallel Programming (Scientific and Engineering Computation), The MIT Press, 2007, ISBN 0262533022.

[4] T.M.R. Ellis, I. R. Phillips, T. M. Lahey, Fortran 90 Programming, Addison Wesley, ISBN-10: 0201544466, 1994.

[5] Lawrence Livermore National Laboratory, OpenMP Tutorial, http://www.llnl.gov/computing/tutorials/OpenMP

[6] Intel Corporation, Parallelization with OpenMP Overview, http://www.intel.com/software/products/compilers/flin/docs/for_ug2/par_opmp.htm

[7] Wikipedia, FORTRAN, http://en.wikipedia.org/wiki/Fortran.