Práctica 4: Programación paralela con OpenMP

Resumen

El objetivo de esta práctica es presentar OpenMP, un modelo de programación paralela en máquinas de memoria compartida. OpenMP permite explicitar las zonas de código paralelo mediante directivas de compilación. Tras describir el entorno de trabajo, se realizará un estudio teórico y práctico que presenta y ejercita el modelo de ejecución de OpenMP y sus directivas básicas. Finalmente se plantean códigos en los que debe analizarse la viabilidad de su ejecución en paralelo y en su caso paralelizar mediante directivas OpenMP.

1. OpenMP

OpenMP [3] es un modelo portable de programación de aplicaciones paralelas en arquitecturas de memoria compartida. Ofrece a los programadores una versátil interfaz para el desarrollo de aplicaciones que se pueden ejecutar en plataformas que varían desde los computadores de sobremesa hasta los supercomputadores.

La interfaz de programación de aplicación (API) en OpenMP soporta programación paralela de memoria compartida en Fortran y C/C++. Diferentes sistemas operativos soportan programas OpenMP (UNIX, Linux, Windows ...).

En la documentación se incluye un extracto de la especificación de la API de OpenMP (versión 3.0, para C/C++ y Fortran) [5].

2. Entorno de trabajo

Los programas OpenMP de las prácticas 4 y 5 se compilarán y ejecutarán en el cluster de prácticas del Departamento de Informática e Ingeniería de Sistemas (DIIS), compuesto por dos máquinas idénticas llamadas hendrix01 y hendrix02. Se trata de dos servidores Sun Enterprise-T5120 con las siguientes características:

- Procesador UltraSPARC-T2[1] de 4 núcleos[2], cada uno capaz de ejecutar 8 threads[3]. Su frecuencia es de 1165 MHz, y tiene 4MB de L2 compartida.
- Memoria RAM: 4 GB

1. También conocido como Niágara 2.
2. Existen versiones de este procesador con 8 núcleos.
3. En el sistema operativo Solaris, el número de procesadores virtuales a los que se les pueden asignar threads para su ejecución puede determinarse con la orden `psrinfo`(1M).

- Disco duro: 2 x 136 GB
- Sistema operativo: Solaris 10 (SunOS 5.10)
- Sun Studio 12 Collection: compiladores C, C++, Fortran y herramientas de desarrollo, depuración y optimización. Su documentación puede consultarse en el siguiente enlace:

  `http://docs.oracle.com/cd/E19205-01/index.html`

Se accede a las máquinas mediante ssh (usando putty en windows), bien conectándose explícitamante a una de las dos (hendrix01 o hendrix02) o bien a hendrix-ssh (lo que conecta a una de las dos aleatoriamente). También puede especificarse hendrix a secas.

En el siguiente enlace puede encontrarse información más detallada del cluster:

`http://diis.unizar.es/WebEstudiantes/hendrix/intro.html`

En caso de problemas de compilación o ejecución, o si se quiere conocer más detalles, consultar la guía del usuario del API de OpenMP (Sun Studio 12: OpenMP API User's Guide):

`http://docs.oracle.com/cd/E19205-01/819-5270/`

3. Estudio teórico-práctico de OpenMP

A continuación se presenta un estudio de OpenMP que se basa en el publicado por el *National Energy Research Scientific Computing Center* (NERSC) [4]. Es un buen material para comprender tanto los modelos de programación y ejecución como las directivas básicas de OpenMP[1].

3.1. Modelo de ejecución

El API de OpenMP usa el modelo *fork-join* de ejecución paralela (ver Figura 1). Un programa OpenMP comienza con un único thread, llamado *master thread*, que ejecuta el código de forma secuencial. Cuando el *master thread* se encuentra con una directiva OpenMP que indica el comienzo de una región paralela, crea un equipo (*team*) de threads y se convierte en el *master* del nuevo equipo. Las instrucciones del programa dentro de la región paralela son ejecutadas en paralelo por cada thread del equipo. Al finalizar la región paralela, los threads del equipo se sincronizan y sólo el *master* thread continúa con la ejecución del programa.

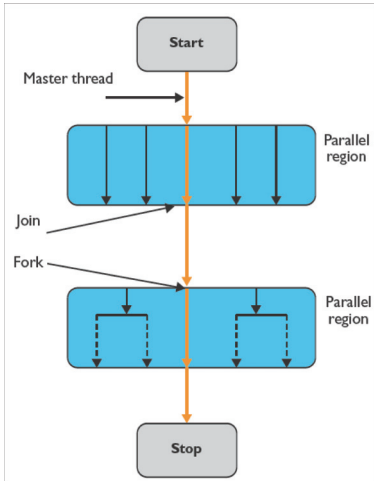

Figura 1 Modelo de ejecución de OpenMP.

El número de threads creados al entrar en una región paralela puede controlarse por medio de una variable de entorno o por una función llamada desde el programa. Dentro de una región paralela puede definirse otra región paralela anidada en la cual cada thread de la región original se convierte en el *master*

1. En caso de dudas al realizar el estudio teórico, o para un conocimiento más avanzado de OpenMP, consultar la especificación de OpenMP [5].

de su propio equipo de threads. El grado de paralelismo de un código OpenMP depende del propio código, del número de procesadores y del sistema operativo. En ningún caso se garantiza que cada thread se ejecute en un procesador diferente.

Las directivas OpenMP, que tienen forma de comentarios Fortran o C/C++, se insertan en puntos clave del código fuente. En caso de que el compilador reconozca las directivas, generará el código necesario para paralelizar la región de código especificada.

3.2. Ejemplos

Los programas que contiene el estudio (Fortran en su mayoría, alguno en C), junto con varios *scripts* con órdenes de compilación y ejecución están en:

```
hendrix://home/chus/pub_code/p41.tar.gz
```

Las órdenes para copiar este fichero en vuestro `$HOME`, descomprimirlo y desagruparlo son:

```
cp /home/chus/pub_code/p41.tar.gz $HOME
gunzip p41.tar.gz
tar xvf p41.tar
```

En general, para cada programa se realizarán las siguientes acciones:

- Compilar con y sin soporte OpenMP (*scripts* `compfomp.sh` y `compfnoomp.sh`, echar un vistazo a las opciones de compilación).
- Ejecutar la versión OpenMP y analizar aspectos de su ejecución (cómo se reparten las iteraciones de los bucles entre los *threads*, la ejecución de las distintas secciones de código ...). Es interesante ejecutar cada programa varias veces o variar el número de threads OpenMP (*script* `ejecuta.sh`, dependiendo del shell que uses, igual tienes que modificarlo). Si es posible, ejecuta también la versión no-OpenMP y compara su salida con la versión OpenMP.

3.3. Directiva parallel

La directiva **parallel** define el inicio de una región paralela. Cuando el *master thread* se encuentra con esta directiva arranca un equipo de threads (*fork*) y cada uno de ellos ejecuta el *code block*. Al finalizar la región paralela (**end parallel**), los threads del equipo se sincronizan y sólo el *master* thread continúa con la ejecución del programa (*join*).

```
!$OMP PARALLEL [clause]

    code block

!$OMP END PARALLEL
```

Entre otros aspectos, `clause` permite especificar si las variables de la región paralela serán privadas para cada thread o compartidas por el equipo.

Es importante recalcar que todo el código de la región paralela se ejecuta por cada uno de los threads del equipo, a menos que otra directiva OpenMP especifique lo contrario. Por ejemplo, un bucle que se encuentre dentro de una región paralela se ejecutará al completo (de forma redundante)

por cada thread a menos que se inserte una directiva **do** antes del bucle. Es necesaria una directiva **do** si se desea que las iteraciones de un bucle se repartan entre los threads de un equipo.

El siguiente código Fortran muestra un ejemplo de uso de la directiva **parallel**:

```
!Filename: parallel.f90

PROGRAM PARALLEL
        IMPLICIT NONE
        INTEGER I

        I=1

!$OMP PARALLEL FIRSTPRIVATE(I)

        PRINT *, I

!$OMP END PARALLEL

END PROGRAM PARALLEL
```

Compila el programa. Puedes hacerlo directamente desde la línea de comandos:

```
hendrix:~/ f90 -xO3 -xopenmp -vpara parallel.f90
```

o con el script `compfomp.sh`.

El número máximo de threads que van a ejecutar un programa compilado para ejecución paralela con OpenMP se controla con la variable de entorno `OMP_NUM_THREADS` (Sun Studio 12: OpenMP API User's Guide [6], pág.13). Su valor por defecto es 1. Las siguientes líneas muestran cómo cambiar el valor de la variable `OMP_NUM_THREADS` a 2 desde bash, ksh y csh:

```
$ export OMP_NUM_THREADS=2 # bash/ksh syntax
% setenv OMP_NUM_THREADS 2 # csh syntax
```

En la página 16 del documento "*Sun Studio 12: OpenMP API User's Guide*" [6] se muestran las variables de entorno que controlan la ejecución de programas OpenMP.

Ejecuta `parallel.f90` con distintos valores de `OMP_NUM_THREADS`. ¿Cuál es el número máximo de threads que soporta?

Compila el programa sin soporte OpenMP, directamente desde la línea de comandos:

```
hendrix:~/ f90 -xO3 parallel.f90[1]
```

o con el script `compfnoomp.sh`. Ejecuta esta versión del código y observa la salida.

1. Si no se especifica -xopenmp en la línea de comandos, el compilador asume -xopenmp=none (reconocimiento de pragmas OpenMP deshabilitado).

3.3.1 Funciones de biblioteca OpenMP

El código `parallel.c` ilustra otro uso de la directiva **parallel** y el empleo de funciones de biblioteca OpenMP para obtener información de los threads en tiempo de ejecución. Compila `parallel.c` (`compc.sh`) y ejecútalo con distintos valores de `OMP_NUM_THREADS`. Observa y analiza la salida.

```
/* parallel.c */
#include <omp.h>
#include <stdio.h>

int main(void)
{
    int i;

    #pragma omp parallel private(i)
    {
        i = omp_get_thread_num();
        if (i == 0)
           printf("soy el master thread (tid = %d)\n", i);
        else
            printf("soy el thread con tid %d\n", i);
    }
    return 0;
}
```

El programa `parallelmal.c` es una variante de este programa que muestra el efecto de definir la variable `i` como compartida.

3.3.2 Paralelismo anidado

Una región paralela OpenMP pueden estar anidada dentro de otra región paralela (por ejemplo, la segunda región paralela de la Figura 1). Si el paralelismo anidado está deshabilitado, entonces el nuevo equipo creado por un thread que se encuentra con una directiva **parallel** dentro de una región paralela se compone sólo por ese mismo thread. Si el paralelismo anidado está habilitado, entonces el nuevo equipo puede componerse por más de un thread.

La biblioteca de ejecución de OpenMP mantiene una reserva (*pool*) de threads que pueden convertirse en threads de un equipo en regiones paralelas. Cuando un thread se encuentra con una directiva parallel y necesita crear un equipo de threads, busca en la citada reserva y se hace con threads ociosos (*idle*), que se convierten en esclavos del equipo. El thread maestro puede obtener menos threads de los que necesita en caso de que no haya suficientes. Cuando el equipo de threads finaliza la ejecución de la sección paralela, los threads esclavos vuelven a la reserva de threads.

El programa `nested.f90` muestra un ejemplo de paralelismo anidado. Compila y ejecuta este programa. Analiza su salida sabiendo que, por defecto, el paralelismo anidado está deshabilitado. Puede activarse cambiando el valor de la variable de entorno `OMP_NESTED` a `TRUE` o quitando el comentario a la línea del programa que llama a la función `OMP_SET_NESTED`. Vuelve a ejecutar el

programa y comprueba la diferencia. Modifica el programa para que varíe el número de threads que ejecutan cada una de las regiones paralelas. ¿Cuál es el número máximo de threads que llegas a ver?

```
!Filename: nested.f90

PROGRAM NESTED
        IMPLICIT NONE
        INTEGER nthreads,tnumber
        INTEGER OMP_GET_NUM_THREADS,OMP_GET_THREAD_NUM

        CALL OMP_SET_DYNAMIC (.TRUE.)
        ! CALL OMP_SET_NESTED (.TRUE.)
        CALL OMP_SET_NUM_THREADS (2)

!$OMP PARALLEL DEFAULT(PRIVATE)

        tnumber=OMP_GET_THREAD_NUM()
        PRINT *, "First parallel region: thread ",tnumber, &
                 " of ", OMP_GET_NUM_THREADS()
        CALL OMP_SET_NUM_THREADS (4)

!$OMP PARALLEL FIRSTPRIVATE(tnumber)

        PRINT *, "Nested parallel region (team ",tnumber, &
               "): thread ",OMP_GET_THREAD_NUM(), &
               " of ", OMP_GET_NUM_THREADS()

!$OMP END PARALLEL

!$OMP END PARALLEL

END PROGRAM NESTED
```

La documentación del API de OpenMP en Sun Studio 12 para Solaris 10 (*Sun Studio 12: OpenMP API User's Guide* [6], pág.40) recoge una serie de pautas de programación relacionadas con el paralelismo anidado.

3.3.3 Reducción

La claúsula (*clause*) **reduction** especifica un operador y una o más variables. Al comenzar la región paralela, se crea una copia privada de cada variable especificada para cada thread y se inicializa apropiadamente según el operador. Al finalizar la región paralela, las variables son actualizadas con los valores de sus copias privadas usando el operador especificado.

El programa `reduction.f90` muestra el uso de **reduction**. Compílalo, ejecútalo con 2, 4 y 8 threads, y analiza su salida.

```
!!Filename: reduction.f90

     PROGRAM REDUCTION
         IMPLICIT NONE
         INTEGER tnumber, OMP_GET_THREAD_NUM
         INTEGER I,J,K

!$OMP PARALLEL DEFAULT(PRIVATE) REDUCTION(+:I)&
!$OMP         REDUCTION(*:J) REDUCTION(MAX:K)
         tnumber=OMP_GET_THREAD_NUM()

         I = tnumber
         J = tnumber
         K = tnumber

         PRINT *, "Thread ",tnumber, "I =",I," J =", J," K =",K

!$OMP END PARALLEL

         PRINT *, ""
         PRINT *, "Thread ",tnumber, "I =",I," J =", J," K =",K=",K

       END
```

3.4. Directiva do

La directiva do especifica que las iteraciones del bucle que se encuentra inmediatamente a continiación deben repartirse entre los distintos threads del equipo. Esta directiva debe estar en una región paralela ya que por sí misma no crea los threads necesarios para ejecutar el bucle en paralelo.

```
!$OMP DO [clause[[,]clause ...]
    do_loop
!$OMP END DO [NOWAIT]
```

Entre otros aspectos, las cláusulas permiten especificar si las variables del bucle serán privadas o compartidas, operaciones de reducción. También puede controlarse la forma de repartir las iteraciones del bucle entre los threads:

- `SCHEDULE(STATIC[,chunk])`: las iteraciones se dividen en bloques del tamaño especificado por *chunk*. Los bloques de iteraciones se asignan a los threads según el algoritmo *round-robin*.
- `SCHEDULE(DYNAMIC[,chunk])`: las iteraciones se dividen en bloques del tamaño especificado por *chunk*. Conforme cada thread finaliza su bloque de iteraciones, se le asigna dinámicamente el siguiente conjunto de iteraciones.
- `SCHEDULE(GUIDED,chunk)`: las iteraciones se dividen en bloques cuyo tamaño decrece de forma exponencial, siendo *chunk* el tamaño más pequeño. Conforme cada thread finaliza su bloque de iteraciones, se le asigna dinámicamente el siguiente conjunto de iteraciones.

- `SCHEDULE(RUNTIME)`: la planificación se retrasa hasta tiempo de ejecución. El tipo de planificación y el tamaño del bloque de iteraciones puede configurarse mediante la variable de entorno `OMP_SCHEDULE`.

Compila el código `dodir.f90`. Ejecútalo con 2 threads y analiza la salida. Verifica que las iteraciones del bucle se han repartido entre los 2 threads de la forma especificada en el código.

```
!Filename: dodir.f90

PROGRAM DODIR
        IMPLICIT NONE
        INTEGER I,L
        INTEGER, PARAMETER:: DIM=16
        REAL A(DIM),B(DIM),S
        INTEGER nthreads,tnumber
        INTEGER OMP_GET_NUM_THREADS,OMP_GET_THREAD_NUM

        CALL RANDOM_NUMBER(A)
        CALL RANDOM_NUMBER(B)

!$OMP PARALLEL DEFAULT(PRIVATE) SHARED(A,B)
!$OMP DO SCHEDULE(STATIC,4)
        DO I=2,DIM
                B(I) = ( A(I) - A(I-1) ) / 2.0
                nthreads=OMP_GET_NUM_THREADS()
                tnumber=OMP_GET_THREAD_NUM()
               print *, "Thread",tnumber," of",nthreads," has I=",I
        END DO
!$OMP END DO
!$OMP END PARALLEL

        S=MAXVAL(B)
        L=MAXLOC(B,1)

        PRINT *, "Maximum gradient: ",S," at location:",L

END PROGRAM DODIR
```

La siguiente tarea es modificar el tamaño de los bloques de iteraciones (*chunks*) que se reparten a los threads. Observa que efectivamente cambia la asignación de iteraciones. Prueba a ejecutar el programa con 8, 16 y 32 threads (tendrás que cambiar el valor de `DIM`).

Por último, comenta las líneas `!$OMP DO` y `!$OMP END DO` para observar que todos threads ejecutan el bucle completo (para comentar una línea, inserta un caracter !).

3.5. Directiva parallel do

La directiva **parallel do** propociona una forma abreviada de especificar una región paralela que contiene un directiva do.

```
!$OMP PARALLEL DO [clause[[,]clause ...]
    do_loop
!$OMP END PARALLEL DO
```

Su semántica es idéntica a la de una directiva **parallel** seguida de una directiva **do**, aunque algunas implementaciones pueden generar códigos diferentes. Por ejemplo, la documentación del API de OpenMP en Sun Studio 12 para Solaris 10 (Sun Studio 12: OpenMP API User's Guide [6], pág.50) indica que la construcción **parallel do** está implementada de forma más eficiente que una región paralela general que puede contener varios bucles.

El código `pardo.f90` ilustra el uso de la directiva **parallel do**. También muestra el empleo de distintas funciones de tiempos. Antes de ejecutar el código, lee con detalle las entradas en el manual (man) de las funciones de tiempos utilizadas en el programa (`dtime`, `etime`, cada una con distinto significado según se trabaje con una o más procesadores). Compila `pardo.f90` y ejecútalo con 1, 2, 4, 8, 16 y 32 threads. Analiza y anota los tiempos de ejecución. Calcula los distintos *speedups* y comenta los resultados.

```
!Filename: pardo.f90

PROGRAM PARDO
         IMPLICIT NONE
         INTEGER I,J
         INTEGER, PARAMETER:: DIM1=50000, DIM2=200
         REAL A(DIM1),B(DIM2,DIM1),C(DIM2,DIM1)
         REAL t1, t2, dtime, etime, S
         INTEGER count1, count2, cr
         REAL tarray1(2),tarray2(2)
         INTEGER nthreadsOMP, tnumber, maxnthreads, numprocsOMP
         INTEGER OMP_GET_NUM_THREADS,OMP_GET_THREAD_NUM
         INTEGER OMP_GET_MAX_THREADS,OMP_GET_NUM_PROCS,
         INTEGER OMP_SET_NUM_THREADS
         INTEGER OMP_SET_DYNAMIC, OMP_GET_DYNAMIC

         CALL RANDOM_NUMBER(A)
         PRINT *," "

         maxnthreads=OMP_GET_MAX_THREADS()
         numprocsOMP=OMP_GET_NUM_PROCS()
         !! CALL OMP_SET_DYNAMIC(.TRUE.)
```



```
         PRINT *, "Entorno de ejecución"
         PRINT *, " - Máximo nº de threads disponibles
                (omp_get_max_threads): ",maxnthreads
         PRINT *, " - Nº de procesadores disponibles
                (omp_get_num_procs): ",numprocsOMP

         call system_clock(count_rate=cr)  ! find count rate
         call system_clock(count=count1) ! start timing
         t2 = etime(tarray2)
         t1 = dtime(tarray1)

!$OMP PARALLEL DO SCHEDULE(RUNTIME) &
!$OMP     PRIVATE(I,J) SHARED (A,B,C,nthreadsOMP)

         DO J=1,DIM2
             nthreadsOMP = OMP_GET_NUM_THREADS()
             DO I=2, DIM1
                 B(J,I) = ( (A(I)+A(I-1))/2.0 ) / SQRT(A(I))
                 C(J,I) = SQRT( A(I)*2 ) / ( A(I)-(A(I)/2.0) )
                 B(J,I) = C(J,I) * ( B(J,I)**2 ) * SIN(A(I))
             END DO
         END DO

 !$OMP END PARALLEL DO

         t1 = dtime(tarray1)
         t2 = etime(tarray2)
         call system_clock(count=count2) ! stop timing

         S=MAXVAL(B)

         PRINT *," - Threads utilizados
                  (omp_get_num_threads)", nthreadsOMP
         PRINT *, " "

         PRINT *,"*** dtime ***"
         WRITE(6,'("Tiempo total=",1pe8.2)') t1
         WRITE(6,'("Tiempo de usuario=",1pe8.2)') tarray1(1)
         WRITE(6,'("Tiempo de sistema=",1pe8.2)') tarray1(2)
         PRINT *, " "
```

```
         PRINT *,"*** etime ***"
         WRITE(6,'("Tiempo total=",1pe8.2)') t2
         WRITE(6,'("Tiempo de usuario=",1pe8.2)') tarray2(1)
         WRITE(6,'("Tiempo de sistema=",1pe8.2)') tarray2(2)
        PRINT *, " "

         PRINT *,"*** system_clock ***"
         WRITE(6,'(" Tiempo total=",1pe8.2)') count2-count1
         PRINT *,"Resolución del reloj del sistema (msg)=", 1.0/cr

 END PROGRAM PARDO
```

3.6. Directiva sections

La directiva **sections** define una región de código que contiene una serie de secciones de código que se deben repartir y ejecutar entre los threads de un equipo. Cada sección es ejecutada una vez por uno de los threads del equipo.

```
!$OMP SECTIONS [clause[[,]clause ...]

[!$OMP SECTION]
    code block
[!$OMP SECTION
    code block]
...

!$OMP END SECTIONS [NOWAIT]
```

Compila el programa `sections.f90`. Ejecútalo con 1, 2, 4 y 8 threads y observa su salida.

```
!Filename: sections.f90

PROGRAM SECTIONS
        IMPLICIT NONE
        INTEGER OMP_GET_THREAD_NUM, tnumber

!$OMP PARALLEL
!$OMP SECTIONS PRIVATE(tnumber)

!$OMP SECTION
        tnumber=OMP_GET_THREAD_NUM()
       PRINT *,"This is section 1 being executed by thread",tnumber
```



```
!$OMP SECTION
        tnumber=OMP_GET_THREAD_NUM()
       PRINT *,"This is section 2 being executed by thread",tnumber
!$OMP SECTION
        tnumber=OMP_GET_THREAD_NUM()
       PRINT *,"This is section 3 being executed by thread",tnumber
!$OMP SECTION
        tnumber=OMP_GET_THREAD_NUM()
       PRINT *,"This is section 4 being executed by thread",tnumber
!$OMP END SECTIONS
!$OMP END PARALLEL

END PROGRAM SECTIONS
```

3.7. Directiva single

La directiva **single** especifica que el código asociado debe ejecutarse solamente por un thread del equipo. Los threads que no ejecutan el código indicado por la directiva single deben esperar en la directiva **end single** a menos que se especifique **nowait**. No está permitido saltar (*branch*) al exterior desde un bloque de código ligado a la directiva **single**.

```
!$OMP SINGLE [clause[[,]clause ...]
    block
!$OMP END SINGLE [NOWAIT]
```

Compila, ejecuta con 4 threads y observa la salida del programa `single.f90`. Los ficheros `fort.10`, `fort.11` y `fort.12` tienen contienen 12 valores (1.0).

```
!Filename: single.f90

PROGRAM SINGLE
        IMPLICIT NONE
        INTEGER, PARAMETER:: N=12
        REAL, DIMENSION(N):: A,B,C,D
        INTEGER:: I
        REAL:: SUMMED

!$OMP PARALLEL SHARED(A,B,C,D) PRIVATE(I)

!***** Reading files fort.10, fort.11, fort.12 in parallel

!$OMP SECTIONS
```

```
!$OMP SECTION
        READ(10,*) (A(I),I=1,N)
!$OMP SECTION
        READ(11,*) (B(I),I=1,N)
!$OMP SECTION
        READ(12,*) (C(I),I=1,N)
!$OMP END SECTIONS

!$OMP SINGLE
        SUMMED = SUM(A) + SUM(B) + SUM(C)
        PRINT *, "Sum of A+B+C=",SUMMED
!$OMP END SINGLE

!$OMP DO SCHEDULE(STATIC,4)
        DO I=1,N
                D(I) = A(I) + B(I)*C(I)
        END DO
!$OMP END DO

!$OMP END PARALLEL

        PRINT *, "The values of D are", D

END PROGRAM SINGLE
```

3.8. Directiva master

El código asociado a la directiva **master** es ejecutado solamente por el thread maestro de un equipo. No hay barreras a la entrada o salida de la dicha sección de código. No está permitido saltar al exterior desde un bloque de código ligado a la directiva **master**.

```
!$OMP MASTER
    block
!$OMP END MASTER
```

3.9. Directiva barrier

La directiva **barrier** sincroniza todos los threads de un equipo. Cuando un thread se encuentra con esta directiva, espera hasta que el resto de threads haya llegado a ese punto.

```
!$OMP BARRIER
```

Compila, ejecuta con 4 threads y observa la salida del programa `barrier.f90`. Repite el mismo proceso tras comentar la línea `!$OMP BARRIER`.


```
!Filename: barrier.f90

PROGRAM ABARRIER
        IMPLICIT NONE
        INTEGER:: L
        INTEGER:: nthreads, OMP_GET_NUM_THREADS
        INTEGER:: tnumber, OMP_GET_THREAD_NUM

!$OMP PARALLEL SHARED(L) PRIVATE(nthreads,tnumber)
        nthreads = OMP_GET_NUM_THREADS()
        tnumber  = OMP_GET_THREAD_NUM()

!$OMP MASTER

        PRINT *, ' Enter a value for L'
        READ(5,*)  L

!$OMP END MASTER

!$OMP BARRIER

!$OMP CRITICAL

        PRINT *, ' My thread number         =',tnumber
        PRINT *, ' Number of threads        =',nthreads
        PRINT *, ' Value of L               =',L
        PRINT *, ''

!$OMP END CRITICAL

!$OMP END PARALLEL

END PROGRAM ABARRIER
```

3.10. Directiva flush

La directiva **flush** marca un punto de sincronización donde se requiere una visión consistente de memoria. La lista de argumentos contiene las variables que deben ser enviadas (`flush`) a memoria para evitar realizar esa tarea con todas las variables del código.

```
!$OMP FLUSH(list)
```

3.11. Directiva atomic

La directiva **atomic** asegura que una dirección específica de memoria sea actualizada atómica-

mente, evitando así su exposición a múltiples escrituras por parte de distintos threads.

```
!$OMP ATOMIC
```

Compila, ejecuta con 4 threads y observa la salida del programa `density.f90`..

```
!Filename: density.f
PROGRAM DENSITY
        IMPLICIT NONE

        INTEGER, PARAMETER:: NBINS=10
        INTEGER, PARAMETER:: NPARTICLES=100000
        REAL:: XMIN, XMAX, MAXMASS, MINMASS

        REAL, DIMENSION(NPARTICLES):: X_LOCATION, PARTICLE_MASS
        INTEGER, DIMENSION(NPARTICLES):: BIN
        REAL, DIMENSION(NBINS):: GRID_MASS, GRID_DENSITY
        INTEGER, DIMENSION(NBINS):: GRID_N

        REAL:: DX,DXINV,TOTAL_MASS,CHECK_MASS
        INTEGER:: I, CHECK_N, XMAX_LOC(1)

        GRID_MASS=0.0
        TOTAL_MASS=0.0
        GRID_N=0
        CHECK_MASS=0.0
        CHECK_N=0

! Initialize particle positions and masses
        CALL RANDOM_NUMBER(PARTICLE_MASS)
        CALL RANDOM_NUMBER(X_LOCATION)

        MAXMASS = MAXVAL(PARTICLE_MASS)
        MINMASS = MINVAL(PARTICLE_MASS)
        XMAX = MAXVAL(X_LOCATION)
        XMIN = MINVAL(X_LOCATION)
        PRINT *, 'MINMASS =',MINMASS,' MAXMASS = ',MAXMASS
        PRINT *, 'XMIN =',XMIN,' XMAX = ',XMAX


! Grid Spacing (and inverse)
        DX = (XMAX-XMIN) / FLOAT(NBINS)
        DXINV = 1/DX
```



```
!$OMP PARALLEL DEFAULT(SHARED) PRIVATE(I) REDUCTION(+:TOTAL_MASS)

!$OMP DO
        DO I = 1, NPARTICLES
                IF (I==XMAX_LOC(1)) THEN
                        BIN(I) = NBINS
                ELSE
                        BIN(I) = 1 + ( (X_LOCATION(I)-XMIN) * DXINV )
                END IF

                IF(BIN(I) < 1 .OR. BIN(I) > NBINS) THEN
!                  Off Grid!
                  PRINT *, 'ERROR: BIN =',BIN(I),' X =',X_LOCATION(I)
                ELSE
!$OMP ATOMIC
                    GRID_MASS(BIN(I)) = GRID_MASS(BIN(I)) &
                              + PARTICLE_MASS(I)
!$OMP ATOMIC
                    GRID_N(BIN(I))    = GRID_N(BIN(I)) + 1

                    TOTAL_MASS = TOTAL_MASS + PARTICLE_MASS(I)
                END IF
        END DO
!$OMP END DO

!$OMP END PARALLEL

        DO I=1, NBINS
                GRID_DENSITY(I) = GRID_MASS(I) * DXINV
        END DO

        PRINT *, 'Total Particles =',NPARTICLES
        PRINT *, 'Total Mass =',TOTAL_MASS
        DO I=1,NBINS
                PRINT *, 'DENSITY(',I,' ) =',GRID_DENSITY(I),' &
                      &MASS(',I,' ) =',GRID_MASS(I)
        END DO

! Check for consistency
        DO I=1,NBINS
                CHECK_MASS = CHECK_MASS + GRID_MASS(I)
                CHECK_N    = CHECK_N + GRID_N(I)
        END DO
```

```
        PRINT *, 'Particles on Grid =', CHECK_N
        PRINT *, 'Total Mass on Grid =', CHECK_MASS

END PROGRAM DENSITY
```

Sin las directivas **atomic**, varios threads podrían tratar de actualizar la variable GRID_MASS al mismo tiempo, causando resultados erróneos. Si se vuelve a ejecutar el programa puede observarse que los valores de `GRID_MASS` y `CHECK_MASS` difieren respecto a la primera ejecución. Esto es debido a que las sumas de números de coma flotante no son completamente asociativas. Recompila y reejecuta el código tras comentar las directivas **atomic**. Observa como falla la conservación de partículas.

3.12. Directiva ordered

El código de un bucle asociado a una directiva **ordered** se ejecuta en el mismo orden que lo haría si las iteraciones se ejecutaran de forma secuencial. La directiva **ordered** sólo puede aparecer en el contexto de una directiva **do** o **parallel do**. No está permitido saltar (*branch*) al exterior desde un bloque de código ligado a la directiva **ordered**.

```
!$OMP ORDERED
    block
!$OMP END ORDERED
```

Compila, ejecuta con 4 threads y observa la salida del programa `ordered.f90`. Repite el mismo proceso tras comentar las directivas **ordered**.

```
! Filename: ordered.f90

PROGRAM ORDERED
        IMPLICIT NONE

        INTEGER, PARAMETER:: N=1000, M=4000
        REAL, DIMENSION(N,M):: X,Y
        REAL, DIMENSION(N):: Z
        INTEGER I,J

        CALL RANDOM_NUMBER(X)
        CALL RANDOM_NUMBER(Y)
        Z=0.0

        PRINT *, 'The first 20 values of Z are:'

!$OMP PARALLEL DEFAULT(SHARED) PRIVATE(I,J)
```



```
!$OMP DO SCHEDULE(DYNAMIC,2) ORDERED
        DO I=1,N
                DO J=1,M
                        Z(I) = Z(I) + X(I,J)*Y(J,I)
                END DO

!$OMP ORDERED
                IF(I<21) THEN
                        PRINT *, 'Z(',I,') =',Z(I)
                END IF
!$OMP END ORDERED

        END DO

!$OMP END DO

!$OMP END PARALLEL

END PROGRAM ORDERED
```

4. Análisis de programas y programación con OpenMP

En los siguientes subapartados se plantean unos códigos que contienen bucles. En todos los casos hay un bucle de inicialización y otro de trabajo. Descartaremos los bucles de inicialización y nos centraremos en los bucles de trabajo. Algunos de estos bucles son susceptibles de ser ejecutados en paralelo y otros tienen dependencias que impiden su paralelización. El trabajo consiste en analizar los bucles y, en aquellos casos que sea posible, insertar las directivas de compilación OpenMP adecuadas. Después se compilarán dichos programas en secuencial y paralelo y finalmente se ejecutarán para verificar la corrección de las directivas insertadas.

Los programas Fortran utilizados en los experimentos y los *scripts* para su compilación están en:

```
hendrix://home/chus/pub_code/p42.tar.gz
```

Las órdenes para copiar este fichero en vuestro `$HOME`, descomprimirlo y desagruparlo son:

```
cp /home/chus/pub_code/p42.tar.gz $HOME
gtar p42.tar.gz
```

4.1. Análisis de dependencias

El análisis de dependencias es una técnica necesaria para extraer paralelismo. Se inspecciona si dos o más referencias acceden a la misma posición de memoria. Si sucede tal hecho, al menos una de las referencias es una escritura, y pertenece a una iteración distinta a la de las lecturas, entonces el bucle no puede ser ejecutado en paralelo (dependencia entre iteraciones).

Teniendo esto en cuenta, analiza los programas `ejer2a`, `ejer2b` y `ejer2c`, indica cuál de sus

bucles principales (no los de inicialización[1]) puede paralelizarse e inserta las directivas OpenMP adecuadas. Comprueba tus respuestas comparando los resultados de las versiones secuencial y paralela de los programas.

```
PROGRAM ejer2a
        integer i
        real a(500), b(500), c(500)

        DO i=1,500
          a(i) = rand()
          b(i) = rand()
          c(i) = 0
        ENDDO

        DO i=1,200
          a(2*i) = b(i)
          c(2*i) = a(2*i)
          c(2*i + 1) = a(2*i + 1)
        ENDDO

        print *,a
        print *,b
        print *,c
END
```

```
PROGRAM ejer2b
        integer i
        real a(400), b(400)

        DO i=1,400
          a(i) = rand()
        ENDDO

        DO i=2,200
          b(i) = a(i) - a(i-1)
        ENDDO

        print *,a
        print *,b
END
```

1. Puede haber dependencias entre llamadas a `rand()`, por lo que los bucles de inicialización no podrán ejecutarse en paralelo.


```
PROGRAM ejer2c
        integer i
        real a(200)

        DO i=1,200
          a(i) = rand()
        ENDDO

        DO i=2,200
          a(i) = a(i) - a(i-1)
        ENDDO

        print *,a
END
```

4.2. Privatización

Existen transformaciones que eliminan las dependencias falsas (dependencias de salida y antidependencias). Por ejemplo, en el código `ejer3a`, todas las iteraciones del bucle principal escriben y leen la variable `t`, así que existen dependencias entre iteraciones (*lcd, loop carried dependences*). Sin embargo, cada iteración utiliza el valor de `t` como almacén de un valor temporal que no se emplea en las demás iteraciones. Esta dependencia puede eliminarse dando a cada iteración una copia de `t`. A esta técnica se le denomina expansión escalar, la dimensión de la variable expandida es igual a la del resto de estructura de datos. Alternativamente, puede indicarse que sólo es necesaria una copia de `t` por thread, esta técnica se denomina privatización, y está soportada por OpenMP.

Considerando lo anterior, analiza los programas `ejer3a` y `ejer3b`, indica qué bucles pueden paralelizarse e inserta las directivas OpenMP adecuadas. Comprueba tus respuestas comparando los resultados de las versiones secuencial y paralela de los programas..

4.3. Sustitución de variables de inducción

La presencia de variables de inducción (variables enteras que son modificadas en cada iteración de un bucle) impide la paralelización de un bucle por dos razones: la variable de inducción es leída y escrita en cada iteración, por tanto genera dependencia entre iteraciones. Además, la referencia a un vector/matriz indexado/a por tales variables impide el correcto análisis del bucle. Para evitar estos problemas, algunos compiladores transforman la variable de inducción y la sustituyen por una expresión dependiente del índice del bucle.

Teniendo esto en cuenta, analiza el programa `ejer4a`, indica si el bucle principal puede paralelizarse e inserta las directivas OpenMP adecuadas. Comprueba tu respuesta comparando los resultados de las versiones secuencial y paralela.

```
PROGRAM ejer3a
        integer i
        real a(200),b(200),c(200)

        DO i=1,200
          a(i) = rand()
        ENDDO

        DO i=1,200
          t = a(i)
          b(i) = t + t**2
          c(i) = t + 2
        ENDDO

        print *,a
        print *,b
        print *,c
END
```

```
PROGRAM ejer3b
        integer i
        real a(200,200),b(200,200),c(200,200),t(200)

        DO i=1,200
            DO j=1,200
              a(i,j) = rand()
              b(i,j) = rand()
              c(i,j) = rand()
            ENDDO
        ENDDO

        DO i=1,200
            DO j=1,200
              t(j) = a(i,j)+ b(i,j)
              c(i,j) = t(j) + c(i,j)
            ENDDO
        ENDDO

        print *,a
        print *,b
        print *,c
END
```



```
PROGRAM ejer4a
        integer i, j
        real a(200),b(200)

        DO i=1,200
          a(i) = rand()
        ENDDO

        j = 0
        DO i=1,100
          j = j + 2
          b(j) = a(i)
        ENDDO

        print *,a
        print *,b
END
```

4.4. Reducción

La reducción opera sobre un vector o matriz y produce una variable de dimensión menor. Una reducción causa dependencias entre iteraciones. En algunos casos, este bucle puede paralelizarse, aunque el nivel dinámico de paralelismo se reducirá exponencialmente.

Analiza el programa `ejer5`, indica si su bucle principal puede paralelizarse e inserta las directivas OpenMP adecuadas. Verifica tu respuesta comparando los resultados de las versiones secuencial y paralela.

5. Bibliografía

[1] Rudolf Eigenmann, Jay Hoeflinger. “Parallelizing and Vectorizing Compilers”. Purdue Univ. School of ECE, High-Performance Computing Lab.ECE-HPCLab-99201, Jan 2000.

Técnicas de análisis y transformación de código para la vectorización y extracción de paralelismo. Incluye una interesante introducción y clasificación de arquitecturas y modelos de lenguajes paralelos. Se entrega con la documentación de la práctica.

[2] William Blume, Ramon Doallo, Rudolf Eigenmann, John Grout, Jay Hoeflinger, Thomas Lawrence, Jaejin Lee, David Padua, Yunheung Paek, Bill Pottenger, Lawrence Rauchwerger, and Peng Tu. "Advanced Program Restructuring for High-Performance Computers with Polaris", (short version of this paper was published in IEEE Computer, December 1996, pp 78-82)

Resumen de técnicas para la extracción automática de paralelismo.

[3] Sitio web oficial de OpenMP: <http://www.openmp.org>

[4] Tutorial de OpenMP: <http://www.nersc.gov/nusers/help/tutorials/openmp/>.

[5] API de OpenMP (versión 3.0): <http://openmp.org/wp/openmp-specifications/>

[6] Sun Studio 12 OpenMP API User’s Guide: <http://docs.oracle.com/cd/E19205-01/819-5270/>

```
      PROGRAM ejer5
              integer i, j
              real a(200),b(200),q

              DO i=1,200
                a(i) = rand()
                b(i) = rand()
              ENDDO

              DO i=1,100
                a(i) = a(i) + b(i)
                q = q + a(i)
              ENDDO

              print *,a
              print *,b
              print *,q
      END
```