CURSO

PROGRAMACIÓN PARALELA UTILIZANDO DIRECTIVAS OpenMP

Organizado por:

Colaboran:

Grupo de Arquitectura de Computadores - UDC

Red Mathematica
Consulting & Computing de Galicia

Cofinanciado por:

Fondo Europeo de Desenvolvemento Rexional
Iniciativa Comunitaria

MINISTERIO DE EDUCACIÓN Y CIENCIA

Programación sobre sistemas de memoria compartida: programación con OpenMP

María J. Martín
Grupo de Arquitectura de Computadores
Universidad de A Coruña
España

Contenidos del curso

- Tema 1: Introducción a la programación paralela
  - Niveles de paralelismo
  - Medidas de rendimiento
  - Tipos de arquitecturas paralelas
  - Paradigmas de programación paralela
  - Programación paralela en el SVGD
  - Compilación y ejecución de programas en el SVGD

Contenidos del curso

- Tema 2: Especificación OpenMP
  - Introducción
  - Compilación y ejecución de programas OpenMP
  - Características principales del estándar OpenMP
  - Directivas para la construcción de paralelismo
  - Directiva THREADPRIVATE
  - Directivas de sincronización
  - Biblioteca de rutinas OpenMP
  - Variables de entorno

Contenidos del curso

- Tema 3: Paralelización a nivel de lazo mediante OpenMP
  - Pasos en la paralelización de un programa
  - Lazos potencialmente paralelos
  - Técnicas de reestructuración de código
  - Consideraciones caché
  - Sobrecarga de la paralelización
  - Casos de estudio
  - Análisis de eficiencia

Contenidos del curso

- Tema 4: Más información
  - Benchmarking
  - Otros compiladores
  - OpenMP 3.0
    - Directiva *task*
    - Cláusula *collapse*
    - Nuevo soporte para el anidamiento de paralelismo
    - Extensión de las formas de planificar un lazo
    - Control portable de *threads*
  - Referencias

Introducción a la programación paralela

Niveles de paralelismo

Computador estándar secuencial:

- Ejecuta las instrucciones de un programa en orden para producir un resultado

Idea de la computación paralela:

- Producir el mismo resultado utilizando múltiples procesadores

Objetivo:

- Obtener un menor tiempo de ejecución

Niveles de paralelismo

Nivel de aplicación dentro de un computador:

- Entre diferentes aplicaciones independientes

Nivel de programa o tarea dentro de una aplicación:

- Entre tareas ejecutables que cooperan

Nivel de procedimiento dentro de un programa:

- Entre fragmentos de código (subrutinas)

Nivel de lazo dentro de un procedimiento:

- Entre iteraciones dentro de un mismo lazo

Niveles de paralelismo

Ejemplo: Paralelismo a nivel de lazo

```
sum=0.
DO i=1,n
  sum=sum+a(i)*b(i)
END DO
```

Código secuencial

```
sum1=0.
DO i=1,n,2
  sum1=sum1+a(i)*b(i)
END DO
sum=sum1+sum2
```

```
sum2=0.
DO i=2,n,2
  sum2=sum2+a(i)*b(i)
END DO
```

Código paralelo

Medidas de rendimiento

Medidas del rendimiento de un programa:

- Aceleración (*Speed-up*): S(p)=T(1)/T(p), p=1,2,...
- Eficiencia: E(p)=(S(p)/p)×100 %

Ejemplo:

Tiempo en 1 CPU = 100 s (=T(1))
Tiempo en 15 CPUs = 12 s (=T(15))

Aceleración = 100/12 = 8.3

Eficiencia = (100/ (12*15))*100 = 56%

Medidas de rendimiento

Ley de Amdahl: La eficiencia obtenida en una implementación paralela viene limitada por la parte secuencial (fracción del programa no paralelizable)

Ejemplo:

Código 20% secuencial:

T(1)=T(paralelo)+T(secuencial)=80+20

Utilizando P procesadores:

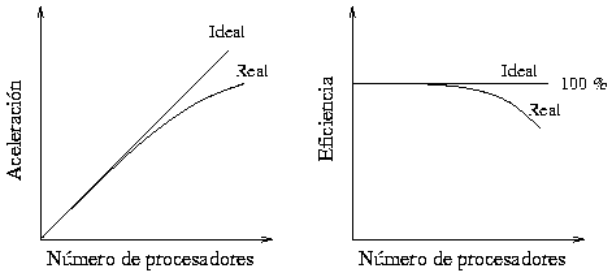
$T(p)=80/p+20 > 20$

$S(p)=T(1)/T(p)=100/(80/p+20) < 5$

$E(p)=S(p)/p < 5/p$ (¡Tiende a CERO!)

Medidas de rendimiento

- El paralelismo tiene un límite
- Si queremos obtener ganancias tenemos que disminuir la parte secuencial ⇒ optimización secuencial



Medidas de rendimiento

- Medidas de tiempo en sistemas Linux:
  - Comando *time:*
  - csh:

```
demo% time a.out
0.04u 0.06s 0:00.51 19.6%
```

Porcentaje de recursos del sistema usados por el programa
Tiempo real
Tiempo de sistema
Tiempo de ejecución


Medidas de rendimiento

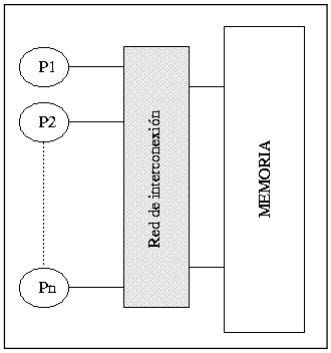
- sh/ksh:

```
demo$ time a.out
real    0m0.14s  ⟶ Tiempo real
user    0m0.00s  ⟶ Tiempo de ejecución
sys     0m0.01s  ⟶ Tiempo de sistema
```

En multiproceso, el tiempo de ejecución dado por “user” es la suma de los tiempos en todos los procesadores ⇒ se debe usar el tiempo real

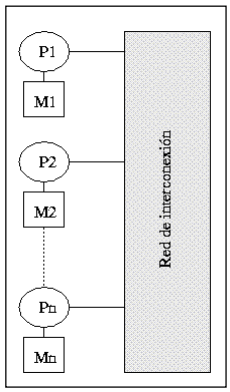

Tipos de arquitecturas paralelas

- **Memoria compartida:** Una memoria única y global accesible desde todos los procesadores (*UMA, Uniform Memory Access*)



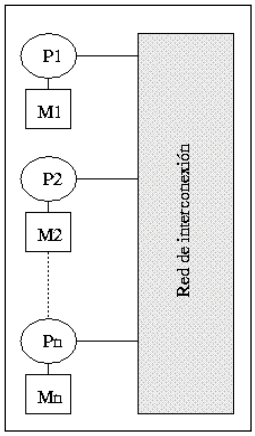
Tipos de arquitecturas paralelas

- **Memoria distribuida:** Tantas memorias locales como procesadores. Un procesador sólo accede a su memoria local.



Tipos de arquitecturas paralelas

- **Memoria compartida-distribuida**: La memoria está físicamente distribuida pero lógicamente compartida (*NUMA, Non-Uniform Memory Access*).



Paradigmas de programación paralela

- **Programación paralela:** Los programas se descomponen en varios procesos que cooperan y se coordinan entre sí
  - **Paradigma de memoria compartida:** La coordinación y cooperación entre los procesos se realiza a través de la lectura y escritura de variables compartidas y a través de variables de sincronización
  - **Paradigma de memoria distribuida:** La coordinación y cooperación entre procesos se realiza mediante el envío y la recepción de mensajes

Paradigma de memoria compartida

- La programación más eficiente, aunque dificultosa, se establece a través de construcciones de bajo nivel (librerías nativas, librería Posix):
  - barreras de sincronización
  - semáforos
  - locks
  - secciones críticas
- En un nivel superior se encuentra la paralelización por medio de directivas. OpenMP es el estándar

Paradigma de memoria compartida

- Paralelización utilizando las librerías del sistema:
  - Tenemos que tener en cuenta todos los detalles de creación de procesos
  - Modifica el código original
- Paralelización utilizando directivas:
  - El programador añade directivas de compilación
  - El compilador los traduce en llamadas a la librería
  - No modifica el código original
  - Más portable y más rápido, pero menos flexible y eficiente

Paradigma de memoria distribuida

- Se utiliza una librería de comunicaciones para intercambio de datos entre los procesadores
- MPI (*Message Passing Interface*) es el estándar

Programación paralela en el SVGD

- SVGD (Superordenador Virtual Gallego):
  - 2 servidores DELL 1425 con procesador Pentium Xeon64 a 3.2 GHz
  - 1 servidor HP Proliant DL145 con 2 procesadores AMD Opteron a 3.2 GHz
  - 18 servidores Pentium III con 36 procesadores a 550 MHz y 800 Mhz
  - 80 servidores DELL 750 con procesadores Pentium a 3.2 GHz
  - **40 servidores blade DELL 1955 con doble procesador quad-core Intel Xeon** (36 Intel Xeon 5310 a 1.6 GHz y 4 Intel Xeon 5355 a 2.66 GHZ)
  - SO: Linux basado en Red Hat Enterprise 4.0

Programación paralela en el SVGD

- Servidores Blade
  - Compiladores disponibles:
    - Lenguaje C/C++: compilador de gnu ***gcc4***
    - Fortran 77/90: compilador de gnu ***gfortran***
  - paradigmas de programación:
    - Memoria compartida (para programación dentro de un nodo): OpenMP disponible
    - Memoria distribuida (para programación entre nodos y dentro de un nodo): MPI disponible

Ejecución de programas en el SVGD

- Ejecución de trabajos en modo interactivo:
  - Una vez conectado al SVG (ssh login@svgd.cesga.es) se abre una sesión interactiva
  - Esta sesión tiene impuesta una serie de límites en cuanto a tiempo de CPU, memoria y disco
  - Los límites se pueden consultar usando el comando Linux:

    demo% ulimit -a

Ejecución de programas en el SVGD

- Ejecución utilizando las colas:
  - El comando para enviar trabajos a colas es:
    demo% qsub –l recurso=valor
  - Lanza el trabajo en la cola que cumple los requisitos impuestos por los recursos pedidos

Ejecución de programas en el SVGD

Recurso	Significado	Unidades	Valor mínimo	Valor máximo
num_proc	Número de CPUs necesarios	Número entero	1	16
s_rt	Máxima cantidad de tiempo real que puede durar el trabajo	Tiempo	00:01:00	300:00:00
s_vmem	Cantidad total de memoria RAM necesaria	Tamaño	112M	4G
h_fsize	Máximo espacio requerido por un único fichero creado por la tarea	Tamaño	1 M	120G
arch	Tipo de procesador en el que se desea ejecutar el trabajo	Valores posibles: 32, 64, opteron, bw		

Ejecución de programas en el SVGD

- Formato de las unidades:
  - Tiempo: especifica el período de tiempo máximo durante el que se puede utilizar un recurso

    [[hh:]mm:]ss

  - Tamaño: especifica el tamaño máximo en Megabytes

    entero[sufijo]

    Sufijo: *K* Kilo (1024) bytes
    M Mega (1048576) bytes
    G Giga (1073741824) bytes


Ejecución de programas en el SVGD

Ejemplos:

demo% qsub –l num_proc=1,s_rt=24:00:00,s_vmem=2G,h_fsize=10G,arch=32
cd $HOME/pruebas
ejecutable
(control)+D

Lee stdin hasta (control)+D

demo%qsub –l num_proc=8,s_rt=40:00,s_vmem=256M,h_fsize=5G job.sh

Usa un script


Ejecución de programas en el SVGD

- Opciones:
  - -cwd: usa como directorio de trabajo el actual (por defecto usa el $(HOME))
  - -o salida: fichero donde se vuelca la salida (por defecto se vuelca a STDIN.ojobid)
  - -e error: fichero donde se vuelca la salida de error (por defecto se vuelca a STDIN.ejobid)
- Ejemplo:

  demo% qsub –cwd –o salida –e /dev/null mi_trabajo.sh

Ejecución de programas en el SVGD

- Como resultado:

  demo%your job 492 (“nombre_ejecutable”) has been submitted

- *qstat* permite conocer el estado del trabajo encolado
  - qstat: muestra el estado de todos los trabajos
  - qstat –u user: muestra los trabajos del usuario user
- *qdel JOBID1 JOBID2 JOBID3 ...* **:** elimina los trabajos listados de la cola

Ejecución de programas en el SVGD

- Entornos de compilación:
  - Para compilar sobre una máquina de la arquitectura adecuada existen los entornos de compilación
  - Los entornos de compilación nos conectan de forma automática a un nodo de la arquitectura seleccionada
  - Comando:
    - *compilar -arch <arquitectura>*
    - Donde <arquitectura> puede ser:
      - 32
      - 64
      - ppteron
      - bw
      - **curso**

Sesión de laboratorio

Lab 1

Especificación OpenMP versión 2

Introducción

- OpenMP es una colección de directivas, librerías y variables de entorno para programas en Fortran, C y C++
- Es actualmente el estándar para programación de sistemas de memoria compartida y sistemas de memoria compartida físicamente distribuida
- Página oficial OpenMP: www.openmp.org

Introducción

- La especificación OpenMP es diseñada por los miembros del OpenMP Architecture Review Board (OpenMP ARB)
- Los miembros actuales del OpenMP ARB son:
  - •DoD ASC
  - •EPCC
  - •Fujitsu
  - •HP
  - •IBM
  - •Intel
  - •KSL
  - •NASA Ames
  - •ST Micro/PG
  - •NEC
  - •Sun
  - •COMPunity
  - •SGI
  - •RWTH Aachen University

Introducción

- La misión de el ARB es estandarizar las APIs para multiprocesamiento de memoria compartida
- Las especificaciones son gratuitas y no requieren licencia
- Historia de las especificaciones OpenMP:
  - 1997: OpenMP 1.0 Fortran
  - 1998: OpenMP 1.0 C/C++
  - 1999: OpenMP 1.1 Fortran
  - 2000: OpenMP 2.0 Fortran
  - 2002: OpenMP 2.0 C/C++
  - 2005: OpenMp 2.5 Fortran, C, C++

Introducción

- OpenMP utiliza el modelo de ejecución paralela fork-join:
  - Un programa comienza su ejecución con un proceso único (thread maestro)
  - Cuando se encuentra la primera construcción paralela crea un conjunto de threads
  - El trabajo se reparte entre todos los threads, incluido el maestro
  - Cuando termina la región paralela sólo el thread maestro continua la ejecución

Introducción

- Normalmente se emplea para la paralelización de lazos:
  - Se busca los lazos computacionalmente más costosos
  - Se reparten sus iteraciones entre los threads

```
void main()
{
  int i;
  double suma[1000];

  for(i=0;i++;i<1000)
    calcular_suma(suma[i]);
}
```

```
void main()
{
  int i;
  double suma[1000];

#pragma omp parallel for
  for(i=0;i++;i<1000)
    calcular_suma(suma[i]);
}
```

Introducción

- Los diferentes threads se comunican a través de variables compartidas
- La compartición de datos puede llevar a un mal comportamiento del programa debido al acceso simultáneo desde diferentes threads
- Para evitarlo se utilizan directivas de sincronización (protegen los datos de conflictos)
- Las sincronizaciones son costosas ⇒ hay que tratar de evitarlas


Ejecución de programas OpenMP en el SVGD

- Los compiladores GNU disponibles en el SVGD soportan la versión 2 de OpenMP para programas Fortran, C y C++
  - opción de compilación -fopenmp
- Más información sobre la implementación GNU de OpenMP en http://gcc.gnu.org/projects/gomp


Ejecución de programas OpenMP en el SVGD

- El número de threads sobre el que se ejecutará el programa paralelo se determina con la variable de entorno OMP_NUM_THREADS

sh/ksh: export OMP_NUM_THREADS=8
csh: setenv OMP_NUM_THREADS 8

estándar

Características principales del estándar

- Sintaxis general de las directivas Fortran:

  centinela nombre_directiva [cláusulas]

- Centinelas:
  - C$OMP, *$OMP (en programas de formato fijo)
  - !$OMP (en programas de formato fijo o variable)
- La directivas comienzan en la columna 1 en formato fijo y en cualquier columna en formato variable
- Las cláusulas pueden ser separadas por espacios o comas. Su orden es indiferente

Características principales del estándar

- La columna 6 debe ser un espacio en la directiva inicial y un ***"&"*** en las líneas de continuación

```
!$OMP PARALLEL DO SHARED(A,B,C)

!$OMP PARALLEL DO
!$OMP&SHARED(A,B,C)
```

- No pueden aparecer directivas OpenMP en procedimientos **PURE** o **ELEMENTAL**

Características principales del estándar

- OpenMP permite la compilación condicional de líneas de programa por medio de centinelas
- La siguiente línea solamente forma parte del programa cuando se compila con OpenMP activado (flag +fopenmp):

```
!$        X(I) = X(I) + XLOCAL
```

Características principales del estándar

- Sintaxis general de las pragmas en C y C++:

#pragma omp nombre_directiva [cláusulas]

- Compilación condicional:

```
#ifdef _OPENMP
X(I)=X(I)+XLOCAL
#endif
```

Características principales del estándar

- En C/C++ tanto las directivas como las cláusulas se escriben siempre con minúsculas
- Fortran acepta tanto minúsculas como mayúsculas

Características principales del estándar

- OpenMP consta básicamente de tres elementos:
  - Control de paralelismo
    - directiva *parallel*
    - directivas de reparto de trabajo (ej, directiva *do)*
  - Control de datos y comunicaciones
    - variables privadas y compartidas
  - Sincronización
    - Para coordinar el acceso a los datos (barreras, secciones críticas ...)

Características principales del estándar

- La mayoría de las directivas se aplican a bloques estructurados de código
- Un bloque estructurado es aquel que no tiene ramas que entren o salgan del bloque

```
10    wrk(id)=garbage(id)
      res(id)=wrk(id)**2
      if(conv(res(id))goto 10

      print*,id
```

bloque estructurado

```
10    wrk(id)=garbage(id)
30    res(id)=wrk(id)**2
      if(conv(res(id))goto 20
      goto 10

      if(not_DONE)goto 10
20    print*,id
```

bloque no estructurado

Directivas para la construcción de paralelismo

Directiva ***PARALLEL:***

- Fortran:

```
!$OMP PARALLEL[cláusulas]
      bloque estructurado
!$OMP END PARALLEL
```

- C/C++:

```
#pragma omp parallel[cláusulas]
    {
     bloque estructurado
     }
```

- Define una región paralela
- Una región paralela es un bloque de código ejecutado en paralelo por varios threads
- El bloque de código debe ser estructurado

Directivas para la construcción de paralelismo

- Cuando un thread encuentra una región paralela :
  - Crea un equipo de threads (número determinado por cláusulas, variables de entorno (***OMP_NUM_THREADS***) o llamadas a la librería (***OMP_SET_NUM_THREADS( )***)
  - Se convierte en el maestro del equipo
  - El thread maestro tiene identificador 0
- Hay una barrera implícita al final de la región paralela
- Al final de la región paralela sólo continúa el thread maestro

Directivas para la construcción de paralelismo

- Cláusulas:
  - IF(expresión escalar lógica)
  - NUM_THREADS(expresión escalar entera)
  - PRIVATE(lista de variables)
  - SHARED(lista de variables)
  - DEFAULT(PRIVATE | SHARED | NONE)
  - FIRSTPRIVATE(lista de variables)
  - REDUCTION(operador: lista de variables)
  - COPYIN(lista de variables)

Directivas para la construcción de paralelismo

- ***IF(expresión escalar lógica):*** Solamente se ejecuta la región paralela si la expresión es cierta
- ***NUM_THREADS(expresión escalar entera):*** El valor de la expresión entera es el número de threads sobre el que se va a ejecutar la región paralela

Directivas para la construcción de paralelismo

- ***PRIVATE(lista):*** Declara privadas a cada thread las variables de la lista.
  - Cada thread tiene una copia local de las variables
  - Una variable local es invisible para los demás threads
  - Las variables no están definidas ni al entrar ni al salir (se utilizan sólo dentro de la región paralela)

Directivas para la construcción de paralelismo

- ***SHARED(lista):*** Declara compartidas las variables de la lista.
  - El valor asignado a una variable compartida es visto por todos los threads del equipo
  - Todos los threads acceden a la misma posición de memoria cuando modifican variables compartidas
  - Puede accederse a la variable concurrentemente
  - No se garantiza la exclusión mutua en el acceso a los datos compartidos (responsabilidad del programador)

Directivas para la construcción de paralelismo

- Se utilizan variables compartidas cuando:
  - tenemos variables de sólo lectura accedidas por diferentes threads
  - los distintos threads acceden a localizaciones diferentes de la variable
  - queremos comunicar valores entre diferentes threads

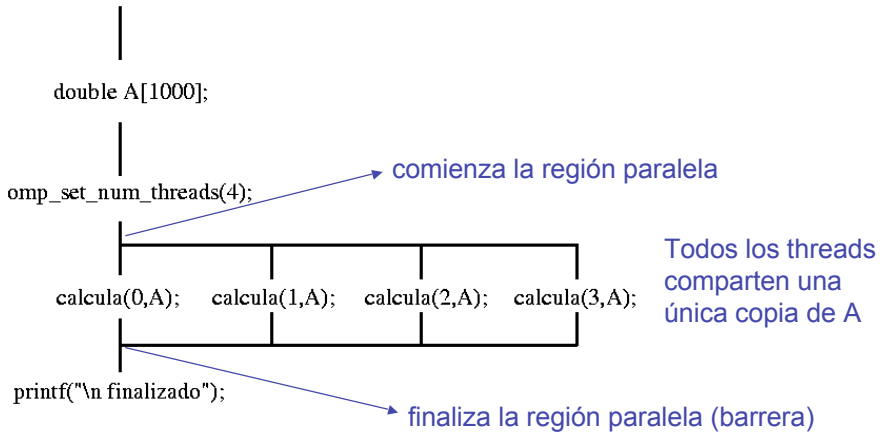
Directivas para la construcción de paralelismo

```
double A[1000];
omp_set_num_threads(4);
#pragma omp parallel private(ID) \
            shared(A)
{
  ID=omp_get_thread_num();
  calcula(ID,A);
}

printf("\n finalizado");
```

- Al llegar a la región paralela se crean 4 threads
- Cada thread ejecuta el código dentro del bloque estructurado
- Se llama a la función *calcula* para id=0,1,2 y 3

Directivas para la construcción de paralelismo



Directivas para la construcción de paralelismo

- ***DEFAULT(PRIVATE | SHARED | NONE):*** Hace que todas las variables dentro de la región paralela sean PRIVADAS, COMPARTIDAS o SIN DEFINIR.
- Sólo la API de Fortran soporta **DEFAULT(PRIVATE)**

Directivas para la construcción de paralelismo

- Si un thread de un equipo ejecutando una región paralela encuentra otra región paralela:
  - crea un nuevo equipo
  - se convierte en el maestro del nuevo equipo
- Por defecto, las regiones paralelas anidadas se serializan (el nuevo equipo está formado por un único thread)
- Este comportamiento se puede cambiar a través de variables de entorno y rutinas de la librería (***OMP_NESTED, OMP_SET_NESTED()***)

Directivas para la construcción de paralelismo

- Directivas de reparto de trabajo (DO/for, SECTIONS, SINGLE)
- Dividen la ejecución de un bloque de código entre los miembros del equipo de threads que la encuentran
- Estas directivas deben ser ejecutadas por todos los miembros del equipo o por ninguno
- Deben estar dentro de una región paralela para que se ejecute en paralelo
- No lanzan nuevos threads
- No hay una barrera implícita en la entrada
- Hay una barrera implícita a la salida

Directivas para la construcción de paralelismo

Directiva ***DO/for*:**

- Fortran:

  !$OMP DO[cláusulas]
  lazo_do
  [!$OMP END DO [NOWAIT]]

- C/C++

  #pragma omp for[cláusulas]
  lazo for

- Indica que las iteraciones del bucle deben ser ejecutadas en paralelo
- No crea nuevos threads, las iteraciones del lazo son repartidas entre los threads que ya existen

Directivas para la construcción de paralelismo

- Existe una barrera implícita al final a menos que se especifique NOWAIT
- Si no se especifica !$OMP END DO se asume que finaliza al finalizar el lazo
- Cláusulas:
  - PRIVATE (lista de variables)
  - FIRSTPRIVATE (lista de variables)
  - LASTPRIVATE (lista de variables)
  - REDUCTION (operador: lista de variables)
  - SCHEDULE (clase[,tamaño del bloque])
  - ORDERED
  - NOWAIT (C/C++)

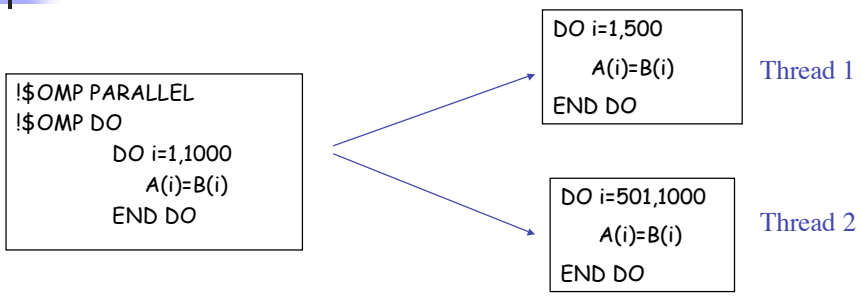
Directivas para la construcción de paralelismo

- Restricciones:
  - La variable de control del lazo debe ser de tipo entero
  - El lazo debe ser un bloque estructurado y no puede usar un ***EXIT*** o un ***GOTO*** (Fortran) ni un ***break*** o un ***GOTO*** (C) para salir del lazo
  - En Fortran los lazos no pueden ser un **DO WHILE** o un **DO** sin control del lazo
  - En C el número de iteraciones del lazo tiene que poder ser computado a la entrada del lazo


Directivas para la construcción de paralelismo

- La mayoría de las variables son compartidas por defecto
- Son privadas:
  - Las variables locales de las subrutinas llamadas desde regiones paralelas
  - Los índices de control de los lazos son privados en Fortran
  - En C/C++ sólo son privados los índices de control de los lazos llamados desde una directiva for


Directivas para la construcción de paralelismo



- El ejecutable da lugar a un número de *threads* determinado.
- Las variables A,B están compartidas, pero i es local a cada *thread* y cogerá un valor inicial y final dependiendo del número de *threads*

Directivas para la construcción de paralelismo

```
        itotal=1000
C$OMP PARALLEL PRIVATE(np, each)
        np=omp_get_num_threads()
        each=itotal/np
        ...
C$OMP END PARALLEL
```

Los dos códigos son equivalentes

```
        itotal=1000
C$OMP PARALLEL DEFAULT(PRIVATE) SHARED(itotal)
        np=omp_get_num_threads()
        each=itotal/np
        ...
C$OMP END PARALLEL
```

Directivas para la construcción de paralelismo

¿Cuál es la diferencia entre estos 2 programas?

```
!$OMP PARALLEL
        DO i=1,10
            print*,"Hola mundo",i
        ENDDO
!OMP END PARALLEL
```

```
!$OMP PARALLEL
!$OMP DO
        DO i=1,10
            print*,"Hola mundo",i
        ENDDO
!OMP END PARALLEL
```

Directivas para la construcción de paralelismo

```
!$OMP PARALLEL DEFAULT (NONE)
!$OMP&SHARED(a,b,c,d) PRIVATE(i)
!$OMP DO
                do i = 1, n-1
                        b(i) = (a(i) + a(i+1))/2
                enddo
!$OMP END DO NOWAIT
!$OMP DO
                do i = 1,n
                        d(i) = 1.0/c(i)
                enddo
!$OMP END DO NOWAIT
!$OMP END PARALLEL
```

Los lazos DO son independientes ⇒ con NOWAIT evitamos esperas innecesarias

Directivas para la construcción de paralelismo

El lazo i es paralelo. ¿Cómo se paraleliza?, ¿qué variables son compartidas y cuales son privadas?

```
maxiter=100
do i=1,m
    do j=1,n
          x=i/real(m)
          y=j/real(n)
          depth(j,i)=calcula(x,y,maxiter)
     enddo
enddo
```

Directivas para la construcción de paralelismo

- ***FIRSTPRIVATE(lista):*** Declara privadas las variables de la lista y además las inicializa a sus valores anteriores (valores en el master).
- ***LASTPRIVATE(lista):*** Declara privadas las variables de la lista y además actualiza las variables con su último valor al terminar el trabajo paralelo.
  - Son variables computadas dentro del lazo y utilizadas fuera (por defecto las variables privadas no existen fuera del lazo)
  - Si una variable se declara como LASTPRIVATE el valor de la variable al salir del bucle es el valor que toma en el procesador que ejecuta la última iteración del bucle

Directivas para la construcción de paralelismo

```
      IS=0
!$OMP PARALLEL PRIVATE(IS)
!$OMP DO
      DO 10 J=1,1000
           IS=IS+J
10    CONTINUE

PRINT*,IS
```

IS está sin inicializar

IS está indefinida en este punto

Directivas para la construcción de paralelismo

```
      IS=0
!$OMP PARALLEL
!$OMP DO FIRSTPRIVATE(IS)
      DO 10 J=1,1000
           IS=IS+J
10    CONTINUE

PRINT*,IS
```

Cada thread consigue su propio IS con valor inicial 0

IS está indefinida en este punto

Directivas para la construcción de paralelismo

```
      IS=0
!$OMP PARALLEL
!$OMP DO FIRSTPRIVATE(IS)
!$OMP+ LASTPRIVATE(IS)
      DO J=1, 1000
           IS=IS+J
10    CONTINUE

PRINT*,IS
```

Cada thread tiene su propio IS con valor inicial 0

IS es definida con su valor en la última iteración

Directivas para la construcción de paralelismo

Warning: Puede producir resultados diferentes a la ejecución secuencial

```
!$OMP PARALLEL
!$OMP DO PRIVATE(I) LASTPRIVATE(X)
          DO I=1,N
             IF(....) THEN
                  X=....
             END IF
          END DO
          PRINT*,X
```

Directivas para la construcción de paralelismo

código secuencial

```
do i=1,n
        z(i)=a*x(i)+y
enddo
```

```
!$OMP PARALLEL
!$OMP DO
        do i=1,n
                z(i)=a*x(i)+y
        enddo
!$OMP END DO
```

código paralelo utilizando una directiva de reparto de trabajo

```
!$OMP PARALLEL
!$OMP&PRIVATE(id,num,istart,iend)
        id=omp_get_thread_num()
        num=omp_get_num_threads()
        istart=id*n/num+1
        iend=min(n,(id+1)*n/num)
            do i=istart,iend
                z(i)=a*x(i)+y
            enddo
!$OMP END PAR
```

código paralelo utilizando una región paralela

Directivas para la construcción de paralelismo

- No todos los lazos son susceptibles de ser paralelizados (problemas de dependencias)
- Si un lazo es paralelo ⇒ no tiene dependencias ⇒ sus iteraciones pueden ser ejecutadas en cualquier orden
- Si un lazo ejecutado en orden inverso no obtiene el mismo resultado ⇒ no es paralelo

Directivas para la construcción de paralelismo

```
DO I=1,N
        A(I)=A(I)+B(I)
ENDDO
```

PARALELO

A(1)=A(1)+B(1)
A(2)=A(2)+B(2)
A(3)=A(3)+B(3)
...
A(N)=A(N)+B(N)

- Cada iteración es independiente del resto
- Se pueden ejecutar en cualquier orden

Directivas para la construcción de paralelismo

```
DO I=1,N
        A(I)=A(I+1)+B(I)
ENDDO
```

NO PARALELO

A(1)=A(2)+B(1)
A(2)=A(3)+B(2)
A(3)=A(4)+B(3)
...
A(N)=A(N+1)+B(N)

- Las iteraciones no se pueden ejecutar en cualquier orden
- La iteración 2 no se puede ejecutar antes que la 1

Directivas para la construcción de paralelismo

```
DO I=2,N
        A(I)=A(I-1)+B(I)
ENDDO
```

NO PARALELO

A(2)=A(1)+B(2)
A(3)=A(2)+B(3)
A(4)=A(3)+B(n)
...
A(N)=A(N-1)+B(N)

- Las iteraciones no se pueden ejecutar en cualquier orden
- La iteración 2 se tiene que ejecutar después de la 1

Directivas para la construcción de paralelismo

¿Son paralelos?

```
DO I=1,N
        A(I)=A(I+N)+B(I)
ENDDO
```

```
DO I=1,N
        A(I)=A(I+M)+B(I)
ENDDO
```

```
DO I=1,N
        C(I)=A(3*I+1)+1
        A(2*I+7)=B(I)-3
ENDDO
```

Directivas para la construcción de paralelismo

```
DO I=1,N
        sum=sum+A(I)
ENDDO
```

NO PARALELO

sum=sum+A(1)
sum=sum+A(2)
sum=sum+A(3)
...
sum=sum+A(N)

- Operación de reducción
- Se puede paralelizar aplicando una transformación

Directivas para la construcción de paralelismo

```
!$OMP PARALLEL PRIVATE(id)
   id=omp_get_thread_num()
   if(id.eq.0)num_pe=omp_get_num_threads()
!$OMP DO
   DO I=1,N
        sum_parcial(id)=sum_parcial(id)+A(I)
   ENDDO
!$OMP END PARALLEL
   DO i=1,num_pe
        sum=sum+sum_parcial(i)
   ENDDO
```

- El compilador es capaz de paralelizar operaciones de reducción
- Solamente hay que indicarle que es una operación de reducción

Directivas para la construcción de paralelismo

- ***REDUCTION(operador:lista):*** Indica una operación de reducción sobre las variables de la lista.
  - Operación de reducción: aplicación de un operador binario conmutativo y asociativo a una variable y algún otro valor, almacenando el resultado en la variable

```
!$OMP PARALLEL
!$OMP DO REDUCTION(+:sum)
    do i=1,n
        sum=sum+a(i)
    enddo
```

```
!$OMP PARALLEL
!$OMP DO REDUCTION (max:x)
    do i=1,n
        x=max(x,a(i))
    enddo
```

Directivas para la construcción de paralelismo

- La cláusula REDUCTION especifica un tipo de dato distinto de SHARED o PRIVATE
- Se puede especificar más de una operación de reducción, pero tienen que ser sobre diferentes variables
- El valor de la variable de reducción es indefinido desde el momento que el primer thread alcanza la cláusula hasta que la operación se completa
- Si se utiliza NOWAIT la variable está indefinida hasta la primera barrera de sincronización

Directivas para la construcción de paralelismo

Operadores de reducción en Fortran		
operador	**tipo de datos**	**valor inicial**
+	entero, flotante (complejo o real)	0
*	entero, flotante (complejo o real)	1
-	entero, flotante (complejo o real)	0
.AND.	lógico	.TRUE.
.OR.	lógico	.FALSE.
.EQV.	lógico	.TRUE.
.NEQV.	lógico	.FALSE.
MAX	entero, flotante (sólo real)	el menor posible
MIN	entero, flotante (sólo real)	el mayor posible
IAND	entero	todos los bits a 1
IOR	entero	0
IEOR	entero	0

Directivas para la construcción de paralelismo

Operadores de reducción en C/C++		
operador	**tipo de datos**	**valor inicial**
+	entero, flotante	0
*	entero, flotante	1
-	entero, flotante	0
&	entero	todos los bits a 1
|	entero	0
^	entero	0
&&	entero	1
||	entero	0

Directivas para la construcción de paralelismo

Precisión numérica en las operaciones de reducción:

- Las operaciones de reducción cambian el orden de las operaciones y por lo tanto el resultado puede ser diferente
- Los errores de redondeo eliminan la propiedad conmutativa de las operaciones

  (-100.0+100.0+1.0e-15)*1.0e+32 = 1.0e+17

  (-100.0+1.0e-15+100)*1.0e+32 = 0.0

Sesión de laboratorio

Lab 2

Directivas para la construcción de paralelismo

- Las cláusulas que determinan el tipo de dato (shared, private, reduction, lastprivate, firstprivate) sólo se aplican a la extensión estática de la región paralela
  - Extensión estática: sentencias delimitadas por la región paralela
  - Extensión dinámica: extensión estática + rutinas llamadas desde la extensión estática


Directivas para la construcción de paralelismo

- OpenMP permite incluir directivas tanto en la extensión estática como en la extensión dinámica del programa
- **Directivas huérfanas:** aquellas que no forman parte de la extensión estática pero sí de la extensión dinámica


Directivas para la construcción de paralelismo

```
#define VECLEN 100
float a[VECLEN], b[VECLEN], sum;

main ()
{
int i;

for (i=0; i < VECLEN; i++)
  a[i] = b[i] = 1.0 * i;

sum = 0.0;
#pragma omp parallel
  sum = dotprod();

printf("Sum = %f\n",sum);
}
```

```
float dotprod ()
{
int i,tid;

tid = omp_get_thread_num();
#pragma omp for reduction(+:sum)
  for (i=0; i < VECLEN; i++)
    {
    sum = sum + (a[i]*b[i]);
    printf("  tid= %d i=%d\n",tid,i);
    }

return(sum);
}
```

Directivas para la construcción de paralelismo

- ***SCHEDULE:*** decide cómo se distribuyen las iteraciones de un lazo entre los threads. La planificación por defecto depende de la implementación.
  - STATIC
  - DYNAMIC
  - GUIDED
  - RUNTIME

El balanceo de la carga entre los procesadores influye de forma importante en el rendimiento

Directivas para la construcción de paralelismo

- ***SCHEDULE(STATIC, chunk):***
  - Distribuye un subconjunto de iteraciones en cada thread de modo circular
  - El tamaño del subconjunto viene dado por *chunk*
  - Si chunk no está especificado la distribución es por bloques
- ***SCHEDULE(DYNAMIC,chunk):***
  - Igual que el anterior pero ahora los bloques son repartidos dinámicamente a medida que los threads van finalizando su trabajo
  - Valor por defecto de chunk = 1


Directivas para la construcción de paralelismo

- ***SCHEDULE(GUIDED,chunk):***
  - Con N iteraciones y P threads distribuye dinámicamente N/kP iteraciones en cada thread (donde k es una constante que depende de la implementación)
  - El valor de N se va actualizando con el valor de las iteraciones que quedan sin asignar hasta un valor mínimo dado por *chunk*
  - La asignación se hace por orden de finalización
  - Valor por defecto de *chunk* = 1
- ***SCHEDULE(RUNTIME):***
  - El tipo de planificación se define en tiempo de ejecución mediante variables de entorno (***OMP_SCHEDULE***)


Directivas para la construcción de paralelismo

Directiva ***SECTIONS***:

- Fortran:

```
C$OMP SECTIONS [cláusulas]
C$OMP SECTION
     bloque estructurado
 C$OMP SECTION
      bloque estructurado
  ...
  C!$OMP END SECTIONS [NOWAIT]
```

Directivas para la construcción de paralelismo

- C/C++:

```
#pragma omp sections [cláusulas]
{
   #pragma omp section
            bloque estructurado
   #pragma omp section
            bloque estructurado
   ...
}
```

Directivas para la construcción de paralelismo

- Permite repartir entre los threads bloques de código independientes (un thread por sección)
- Los bloques de código deben ser estructurados
- Existe una barrera implícita al final de la ejecución de las secciones a no ser que se especifique NOWAIT
- Cláusulas:
  - PRIVATE
  - FIRSTPRIVATE
  - LASTPRIVATE
  - REDUCTION
  - NOWAIT (C/C++)

Directivas para la construcción de paralelismo

```
          integer n
          real a(n)

!$OMP PARALLEL
!$OMP SECTIONS
!$OMP SECTION
          call calcular_media(a,n)
!$OMP SECTION
          call calcular_maximo(a,n)
!$OMP SECTION
          call calcular_desviacion(a,n)
!$OMP END  SECTIONS
!$OMP END PARALLEL
```

Directivas para la construcción de paralelismo

Directiva ***SINGLE:***

- Fortran:

!$OMP SINGLE [cláusulas]
bloque estructurado
!$OMP END SINGLE[NOWAIT]

- *C/C++:*

#pragma omp single [cláusulas]
bloque estructurado


Directivas para la construcción de paralelismo

- El bloque de código es ejecutado por un único thread del equipo
- Los demás threads esperan (barrera implícita al final) salvo que se especifique lo contrario (NOWAIT)
- El bloque de código tiene que ser estructurado
- Cláusulas:
  - PRIVATE
  - FIRSTPRIVATE
  - COPYPRIVATE
  - NOWAIT (C/C++)


Directivas para la construcción de paralelismo

- ***COPYPRIVATE(lista de variables):*** El valor de la/las variables privadas son copiadas a la/las variables privadas de los otros threads al final de la directiva SINGLE.

El uso de esta cláusula es incompatible con el uso de NOWAIT

Directivas para la construcción de paralelismo

```
#pragma omp parallel
        hacer_cosas();
#pragma omp single
 {
        leer_datos();
 }
        hacer_mas_cosas();
#pragma omp single
{
        escribir_resultados();
}
```

Directivas para la construcción de paralelismo

sólo Fortran

Directiva ***WORKSHARE:***

- Fortran:

```
!$OMP WORKSHARE
    bloque estructurado
!$OMP END WORKSHARE [NOWAIT]
```

- Permite la paralelización de expresiones con arrays en sentencias Fortran 90
- Divide las tareas asociadas al bloque en unidades de trabajo y las reparte entre los threads
- El reparto depende del compilador. Única restricción: cada unidad de trabajo se ejecuta una única vez
- Barrera implícita al final salvo que se especifique lo contrario (NOWAIT)

Directivas para la construcción de paralelismo

- Válido también para funciones intrínsecas que operan con arrays (matmul, dot_product, sum, product, maxval, minval, count ...) aunque el rendimiento dependerá del compilador.

sólo Fortran

```
    real, dimension(n,n)::a,b,c
    ...
!$OMP PARALLEL
   ...
!$OMP WORKSHARE
    c=a+b
!$OMP END WORKSHARE
!$OMP END PARALLEL
```

Directivas para la construcción de paralelismo

- Directivas combinadas:
  - ***PARALLEL DO/ parallel for***
    - Es la abreviatura para una región paralela que sólo tiene un lazo.
    - Fortran:
      - Si no se especifica !$OMP END PARALLE DO se asume que la región paralela finaliza cuando finaliza el lazo DO.
      - No se puede utilizar NOWAIT
    - Cláusulas: Cualquiera de las aceptadas por PARALLEL y DO/for excepto nowait (C/C++)

Directivas para la construcción de paralelismo

- ***PARALLEL SECTIONS***
  - Es la abreviatura para una región paralela que contiene sólo una directiva SECTIONS
  - Cláusulas: Cualquiera de las aceptadas por PARALLEL y SECTIONS excepto NOWAIT
- ***PARALLEL WORKSHARE*** (sólo Fortran)
  - Es la abreviatura para una región paralela que contiene sólo una directiva WORKSHARE
  - Cláusulas: Cualquiera de las aceptadas por PARALLEL
  - No se puede utilizar NOWAIT

Directivas para la construcción de paralelismo

```
        real, dimension(n,n):: b,c
        real, dimension(n)::d
        ....
!$OMP PARALLEL WORKSHARE REDUCTION(+:d)
        d(1:100)=d(1:100)+c(1:100,1)*b(1,1:100)
!$OMP END PARALLEL WORKSHARE
```

Sesión de laboratorio

Lab 3

Directiva THREADPRIVATE

- Fortran:

  !$OMP THREADPRIVATE(lista de variables)

- *C/C++:*

  #pragma omp threadprivate(lista de variables)

Directiva THREADPRIVATE

- Esta directiva declara a las variables de la lista privadas a cada thread pero globales dentro del thread
- La directiva THREADPRIVATE debe ser especificada cada vez que la variable es declarada
  - En el caso de bloques common la directiva debe aparecer justo debajo del bloque common y cada vez que aparezca el bloque
  - En el caso de variables estáticas debe aparecer antes de cualquier referencia a las variables

Directiva THREADPRIVATE

- Los datos THREADPRIVATE serán indefinidos dentro de la región paralela a menos que se use la cláusula COPYIN en la directiva PARALLEL
- ***COPYIN(lista de variables):*** Las variables toman el valor que tienen en el master en todos los threads


Directiva THREADPRIVATE

```
            integer istart, iend
            common /bounds/ istart, iend
!$OMP THREADPRIVATE (/bounds/)
            integer datos(1000),N,nthread,id,chunk

            N=1000
!$OMP PARALLEL PRIVATE(id,nthreads,chunk)
            nthreads=omp_get_num_threads()
            id=omp_get_thread_num()
            chunk=(N+nthreads-1)/nthreads
            istart=id*chunk+1
            iend=min((id+1)*chunk,N)
            call work(datos)
!$OMP END PARALLEL
            end
```

```
            subroutine work(datos)
            common /bounds/ istart,iend
!$OMP THREADPRIVATE(/bounds/)
            integer datos(1000),i

            do i=istart,iend
               datos(i)=i*1
            enddo
            return
            end
```


Directiva THREADPRIVATE

```
int istart, iend;
#pragma omp threadprivate(istart,iend)

main()
{
  int datos(1000)

  N=1000;
  #pragma parallel private(id,nthreads,chunk)
        {
            nthreads=omp_get_num_threads();
            id=omp_get_thread_num();
            chunk=(N+nthreads-1)/nthreads;
            istart=id*chunk;
            iend=min((id+1)*chunk,N);
            work(datos);
         }
```

```
void work(datos)
{
  int datos(1000);
  for(i=istart;i<iend;i++)
       datos(i)=i*1;
}
```

Directivas de sincronización

Directiva ***MASTER***

- Fortran:

  !$OMP MASTER

  bloque estructurado

  !$OMP END MASTER

- C/C++:

  #pragma omp master

  bloque estructurado

- Sólo el maestro del grupo ejecuta el bloque de código
- El resto de threads saltan el bloque de código y continúan la ejecución
- No hay barreras implícitas ni al inicio ni al final

Directivas de sincronización

Directiva ***CRITICAL***

- Fortran:

```
!$OMP CRITICAL [(nombre)]
      bloque estructurado
!OMP END CRITICAL [(nombre)]
```

- C/C++:

```
#pragma omp critical [(nombre)]
   bloque estructurado
```

Directivas de sincronización

- Implementa una sección crítica: Todos los threads ejecutan el bloque pero sólo un thread al mismo tiempo tiene acceso al código delimitado por esta directiva
- Un thread espera al inicio de la sección crítica hasta que ningún otro thread está ejecutando una sección crítica con el mismo nombre

Directivas de sincronización

```
!$OMP PARALLEL DEFAULT(PRIVATE), SHARED(X)
!$OMP CRITICAL(XAXIS)
        CALL GET_WORK(IX_NEXT,X)
!$OMP END CRITICAL(XAXIS)
        CALL WORK(IX_NEXT,X)
!$OMP END PARALLEL
```

Cada thread accede a una cola de tareas para conseguir el siguiente trabajo a ejecutar.

Directivas de sincronización

Directiva ***BARRIER***

- Fortran:
  !$OMP BARRIER
- C/C++:
  #pragma omp barrier

- Sincroniza a todos los threads de un grupo
- Espera hasta que todos los threads han alcanzado este punto del programa

Directivas de sincronización

Directiva ***ATOMIC***

- Fortran:
  !$OMP ATOMIC
- C/C++:
  #pragma omp atomic

- Se aplica a la sentencia inmediatamente posterior
- Asegura que una localización de memoria específica va a ser actualizada de forma atómica (indivisible) ⇒ no se van a producir múltiples escrituras desde diferentes threads
- Es una optimización de la sección crítica (dependiendo de la implementación)


Directivas de sincronización

- La sentencia en Fortran para la directiva ATOMIC debe ser de la forma:
  x = x operador *expr*
  x = *expr* operador x
  x = intrínseca (x, *expr*)
  x = intrínseca (*expr*, x)

  operador: +, *, -, /, .AND., .OR., .EQV., o .NEQV.
  intrínseca: MAX, MIN, IAND, IOR o IEOR


Directivas de sincronización

- La sentencia en C/C++ para la directiva atomic debe ser de la forma:

  x binop = expr

  x++

  ++x

  x--

  --x

  binop: +, +, -, /, &, ^, |, >>, <<

Directivas de sincronización

```
!$OMP PARALLEL DO DEFAULT(PRIVATE)
!$OMP&SHARED(X,Y,INDEX,N)
        DO I=1,N
                CALL WORK(XLOCAL,YLOCAL)
!$OMP ATOMIC
                X(INDEX(I)) = X(INDEX(I)) + XLOCAL
                Y(I) = Y(I) + YLOCAL
ENDDO
```

Directivas de sincronización

Directiva ***FLUSH:***

- Fortran:
  !$OMP FLUSH [(lista de variables)]
- C/C++:
  #pragma omp flush [(lista de variables)]

- Es un punto de encuentro que permite asegurar que todos lo threads de un equipo tienen una visión consistente de ciertas variables de memoria

Directivas de sincronización

- Asegura que las modificaciones hechas por un thread sobre variables visibles por otros threads son efectivas (se han almacenado en memoria y se trabaja con estos valores y no con los valores almacenados en los registros)
- Un FLUSH proporciona consistencia entre operaciones de memoria del thread actual y la memoria global
- Para alcanzar consistencia global cada thread debe ejecutar una operación FLUSH
- Si no se especifica una lista de variables se aplica a todas (podría ser muy costoso)

Directivas de sincronización

- Está presente implícitamente en las siguientes directivas (salvo que esté presente la directiva NOWAIT):
  - BARRRIER,
  - CRITICAL y END CRITICAL,
  - ORDERED y END ORDERED
  - PARALLEL y END PARALLEL,
  - END DO,
  - END SECTIONS,
  - END SINGLE,
  - END WORKSHARE,
  - PARALLEL DO/for y END PARALLEL DO/for,
  - PARALLEL SECTIONS y END PARALLEL SECTIONS,
  - PARALLEL WORKSHARE y END PARALLEL WORKSHARE
  - ATOMIC

Directivas de sincronización

```
!$OMP PARALLEL DEFAULT(PRIVATE), SHARED(sinc)
        id=omp_get_thread_num()
        vecino=id+1
        sinc(id)=0
!$OMP BARRIER
        call work()
        sinc(id)=1
!$OMP FLUSH(sinc)
        DO WHILE(sinc(vecino).EQ.0)
!$OMP FLUSH(sinc)
        ENDO
!$OMP END PARALLEL
```

Directivas de sincronización

Directiva ***ORDERED***

- Fortran

```
!$OMP ORDERED
      bloque estructurado
!$OMP END ORDERED
```

- C/C++

```
#pragma omp ordered
   bloque estructurado
```

- Se utiliza dentro de lazos. La directiva DO/for o PARALLEL DO/parallel for origen tienen que incluir la cláusula ORDERED
- Las iteraciones del lazo deben ejecutarse en el mismo orden en el que se ejecutaría en el código secuencial en el bloque delimitado por la directiva


Directivas de sincronización

- Esta directiva secuencializa y ordena el código dentro de la sección ordenada a la vez que permite ejecutar en paralelo el resto

```
!$OMP PARALLEL DO ORDERED
                  DO I=1,N
                     CALL WORK(A(I))
!$OMP ORDERED
                     WRITE(*,*) A(I)
!$OMP END ORDERED
                  ENDDO
```


Biblioteca de rutinas OpenMP

- Para utilizar las funciones en Fortran tenemos que incluir el fichero **omp_lib.h** o el módulo de Fortran 90 **omp_lib**:

  INCLUDE "omp_lib.h"

  o

  USE OMP_LIB
- Para utilizar las funciones en C/C++ tenemos que incluir el fichero de cabecera **<omp.h>:**

  #include <omp.h>
- Tenemos tres tipos de funciones:
  - funciones que modifican el entorno de ejecución
  - funciones de sincronización
  - funciones para medir tiempos

Biblioteca de rutinas OpenMP

OMP_SET_DYNAMIC(expr_log_esc.):

- Habilita o deshabilita el ajuste dinámico del número de threads en las regiones paralelas
- Tiene que ser llamada desde una región secuencial
- Si está habilitada, el número de threads especificado por el usuario (a través de la cláusula NUM_TREADS, la función OMP_SET_NUM_THREADS, o la variable de entorno OMP_NUM_THREADS) pasa a ser un límite máximo
- El número de threads siempre permanece constante dentro de cada región paralela y su número se puede obtener mediante la función OMP_GET_NUM_THREADS( )
- La función tiene prioridad sobre la variable de entorno OMP_DYNAMIC

Biblioteca de rutinas OpenMP

- La especificación OpenMP no obliga a proporcionar implementación del ajuste dinámico aunque la función debe existir por motivos de portabilidad
- El valor por defecto depende de la implementación


Biblioteca de rutinas OpenMP

función lógica OMP_GET_DYNAMIC( ):

- Devuelve .TRUE. si el ajuste dinámico del número de threads está habilitado y .FALSE. en otro caso

OMP_SET_NUM_THREADS(expresión entera):

- Tiene que ser llamada desde una región secuencial
- Especifica el número de threads a usar con la siguiente región paralela
- Tiene prioridad sobre la variable de entorno OMP_NUM_THREADS pero no sobre la cláusula NUM_THREADS
- Cuando se utiliza el ajuste dinámico de threads esta subrutina establece el número máximo de threads posible


Biblioteca de rutinas OpenMP

función entera OMP_GET_NUM_THREADS( ):

- Devuelve el número de threads del equipo ejecutando la región paralela desde la cual es llamada

función entera OMP_GET_MAX_THREADS( ):

- Devuelve el número máximo de threads que puede ser utilizado en la región paralela

función entera OMP_GET_THREAD_NUM():

- Devuelve el número del thread dentro del equipo (valor entre 0 y OMP_GET_NUM_THREADS( ) -1)

función entera OMP_GET_NUM_PROCS( ):

- Devuelve el número de procesadores disponibles

Biblioteca de rutinas OpenMP

```
!$OMP PARALLEL DEFAULT(PRIVATE), SHARED(X,NPOINTS)
        IAM = OMP_GET_THREAD_NUM( )
        NP = OMP_GET_NUM_THREADS( )
        IPOINTS = NPOINTS/NP
        CALL SUBDOMAIN(X,IAM,IPOINTS)
!$OMP END PARALLEL
```

Cada thread trabaja sobre una parte diferente del array X

Biblioteca de rutinas OpenMP

función lógica OMP_IN_PARALLEL( ):

- Devuelve .TRUE. si es llamada desde la extensión dinámica de una región ejecutándose en paralelo y .FALSE. en caso contrario

OMP_SET_NESTED(expr_log_esc):

- Tiene que ser llamada desde una región secuencial
- Habilita o deshabilita el anidamiento de paralelismo (deshabilitado por defecto)
- Tiene prioridad sobre la variable de entorno OMP_NESTED
- El número de threads usado para ejecutar una región paralela anidada depende de la implementación (pudiendo ser 1)

Biblioteca de rutinas OpenMP

función lógica OMP_GET_NESTED( ):

- Devuelve .TRUE. si el anidamiento de paralelismo está habilitado y .FALSE. en otro caso.

OMP_INIT_LOCK(var):

- Inicializa un LOCK asociado a la variable *var*
- Su estado inicial es UNLOCKED

OMP_DESTROY_LOCK(var):

- Elimina el LOCK asociado a *var*

- En C/C++ la variable lock debe ser del tipo omp_lock_t. Todas las funciones lock requieren como argumento un puntero al tipo omp_lock_t
- En Fortran la variable lock es tipo integer*8

Biblioteca de rutinas OpenMP

OMP_SET_LOCK(var) y OMP_SET_NEST_LOCK(var):

- Hace que el thread espere hasta que la variable LOCK especificada esté disponible
- Cuando esté disponible, el thread se hace propietario del LOCK (estado LOCKED)
- La variable LOCK tiene que haber sido previamente inicializada

OMP_UNSET_LOCK(var) y OMP_UNSET_NEST_LOCK(var):

- Libera al thread de la propiedad del LOCK (estado UNLOCKED).

Biblioteca de rutinas OpenMP

función lógica OMP_TEST_LOCK(var):

- Esta rutina intenta activar el LOCK asociado a la variable *var* devolviendo .TRUE. si la operación tuvo éxito y .FALSE. en otro caso

función real OMP_GET_WTIME():

- Devuelve el tiempo de ejecución en segundos hasta ese punto del programa

Biblioteca de rutinas OpenMP

```
          PROGRAM USANDO_LOCKS
          EXTERNAL OMP_TEST_LOCK
          LOGICAL OMP_TEST_LOCK
          INTEGER LCK, ID

          CALL OMP_INIT_LOCK(LCK)
!$OMP PARALLEL SHARED(LCK), PRIVATE(ID)
          ID = OMP_GET_THREAD_NUM( )
          CALL OMP_SET_LOCK(LCK)
                    PRINT*,"Mi ID es", ID
          CALL OMP_UNSET_LOCK(LCK)

          DO WHILE(OMP_TEST_LOCK(LCK).EQV.FALSE.)
                    CALL DUMMY_WORK(ID)
          ENDDO

          CALL DO_WORK(ID)
          CALL OMP_UNSET_LOCK(LCK)
!$OMP END PARALLEL
          CALL OMP_DESTROY_LOCK(LCK)
          STOP
          END
```

Biblioteca de rutinas OpenMP

```
          SUBROUTINE DUMMY_WORK(ID)
          INTEGER ID

          PRINT*, "Thread ID " ID, "esta esperando por el LOCK"
          RETURN
          END

          SUBROUTINE DO_WORK(ID)
          INTEGER ID,I,N

          IF(TID.NEQ.0)THEN
                    N = 100
          ELSE
                    N = 10
          ENDIF

          PRINT*, "Thread ID ", ID, "N = ", N
          RETURN
          END
```

Biblioteca de rutinas OpenMP

- Dentro de un programa C: double omp_get_wtime(void);

```
#include <omp.h>

main()
{
  double start,end;


     start = omp_get_wtime();
     ... trabajo a medir ...
     end = omp_get_wtime();

     printf("Tiempo= %f segundos \n", (end - start));
}
```

Biblioteca de rutinas OpenMP

- Dentro de un programa Fortran:

  DOUBLE PRECISION OMP_GET_WTIME()

```
PROGRAM  MAIN
INCLUDE "omp_lib.h"

DOUBLE PRECISION  start, end;

    start = OMP_GET_WTIME()
    ... trabajo a medir ...
    end = OMP_GET_WTIME()

    print*,'Tiempo= ', (end - start),' segundos'
END
```

Variables de entorno

OMP_SCHEDULE:

- El valor por defecto depende de la implementación
- Sólo aplicable a lazos paralelos en los que se haya especificado una planificación RUNTIME por medio de la directiva SCHEDULE
- Se puede determinar tanto el tipo de planificación como el tamaño del *chunk*
- Si no se especifica *chunk* se toma el valor 1 por defecto (excepto en el caso *static* que se realiza una distribución por bloques)
- Ejemplos:

```
export OMP_SCHEDULE="GUIDED,4"
setenv  OMP_SCHEDULE "dynamic"
```

Variables de entorno

OMP_NUM_THREADS:

- El valor por defecto depende de la implementación
- Especifica el número de threads a usar en una región paralela
- Su valor puede ser cambiado por la subrutina OMP_SET_NUM_THREADS( ) o la cláusula NUM_THREADS
- Si el ajuste dinámico del número de threads está habilitado, el valor de esta variable especifica el número máximo de threads a usar

Variables de entorno

OMP_DYNAMIC:

- El valor por defecto depende de la implementación
- Habilita o deshabilita el ajuste dinámico del número de threads
- Ejemplo:
  export OMP_DYNAMIC=true
- Su valor puede ser cambiado por la subrutina OMP_SET_DYNAMIC( )


Variables de entorno

OMP_NESTED:

- Habilita o deshabilita el anidamiento de paralelismo
- Ejemplo:
  setenv OMP_NESTED TRUE
- El valor por defecto es *FALSE*
- *Su valor puede ser cambiado por la subrutina OMP_SET_NESTED( )*


Sesión de laboratorio

Lab 4

Paralelización a nivel de lazo mediante OpenMP

Pasos en la paralelización de un programa

- La mayoría de las aplicaciones científicas y de ingeniería consumen la mayor parte del tiempo de ejecución en la ejecución de lazos
- Normalmente será suficiente explotar el paralelismo a nivel de lazo para conseguir un rendimiento aceptable


Pasos en la paralelización de un programa

1. Partir del **programa serie optimizado** y obtener resultados para validar
2. Localizar el lazo secuencial más costosos
3. ¿Es el lazo del paso 2 paralelo?
   - 3.1 Si ⇒ Insertar las directivas necesarias
   - 3.2 No ⇒ Modificar el código para hacerlo paralelo e insertar las directivas necesarias
4. Verificar que los resultados son iguales que en la versión serie
5. Medir el rendimiento: Comprobar la aceleración para un número creciente de procesadores (cuidado con el planificador empleado).
6. ¿Son los resultados obtenidos satisfactorios?
   - 6.1 Si ⇒ Fin del proceso de paralelización
   - 6.2 No ⇒ Volver al paso 2


Lazos potencialmente paralelos

- Un lazo es potencialmente paralelo si:
  - No existen dependencias de datos
  - En Fortran:
    - Es un lazo DO (no se pueden paralelizar los DO WHILE)
    - No puede contener setencias exit o gotos que salten fuera del lazo
  - En C:
    - Es una lazo for
    - No puede contener sentencias break ni gotos que dejen el lazo

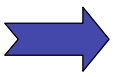
Técnicas de reestructuración de código

Variable de inducción

```
INTEGER  m,k,i
REAL  x(4000), y(1000)

k=4
m=0
do i=1,1000
  m=m+k
  x(m)=y(i)
end do
```

SERIE



```
INTEGER  m,k,i
REAL  x(4000), y(1000)

k=4
!$OMP PARALLEL  DO PRIVATE(m,i)
!$OMP& SHARED(x,y,k)
do i=1,1000
  m=i*k
  x(m)=y(i)
end do
m=1000*k
```

PARALELO

Técnicas de reestructuración de código

```
double up =1.1;
double Sn=1000.0;
double opt[N+1];
int n;

for(n=0;n<=N;++n){
  opt[n]=Sn;
  Sn*=up;
}
```

SERIE

```
double up =1.1;
double Sn, oldSn=1000.0;
double opt[N+1];
int n;

opt[0]=origSn;
#pragma omp parallel for prvate(n) shared(opt)
lastprivate(Sn)
for(n=1;n<=N;++n){
  Sn=oldSn*pow(up,n-1);
  opt[n]=Sn;
}
```

PARALELO

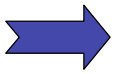
Técnicas de reestructuración de código

Recurrencia hacia atrás

```
INTEGER i
REAL  x(1001)

do i=1,1000
  x(i)=x(i+1)
end do
```

SERIE



```
INTEGER i
REAL  x(1001), xold(1001)

do i=1,1000
  xold(i)=x(i+1)
end do
do i=1,1000
  x(i)=xold(i)
end do
```

PARALELO

Técnicas de reestructuración de código

División del lazo (fisión)

```
INTEGER  i,n,k
REAL a(1000), b(1000), c(1000), f(2000), d

do i=2,n
  a(i)=b(i)+c(i)*d+f(i+k)
  c(i)=a(i-1)
end do
```

SERIE

```
INTEGER  i,n,k
REAL a(1000), b(1000), c(1000), f(2000), d

do i=2,n
  a(i)=b(i)+c(i)*d+f(i+k)
end do
do i=2,n
  c(i)=a(i-1)
end do
```

PARALELO

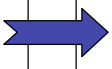
Técnicas de reestructuración de código

Sacar fuera la parte serie

```
      INTEGER a(1000), n, predicado, i

      do i=1,n
        if(predicado(a(i)).eq.1) then
          call transformar(a(i))
        else
          go to 2
        end if
      end do
2     continue
```

SERIE



```
      INTEGER a(1000), n, predicado, i, itemp

      do i=1,n
        if(predicado(a(i)).neq.1)  then
          itemp=i
          go to 2
        end if
      end do
2     continue
!$OMP PARALLEL  DO PRIVATE(i)
!$OMP&SHARED(a,itemp)
      do i=1,itemp-1
          call transformar(a(i))
      end do
```

PARALELO

Consideraciones caché

- Muy habitualmente la memoria limita el rendimiento de los programas de memoria compartida
- Para alcanzar buenos rendimientos la localidad de los datos es un elemento esencial


Consideraciones caché

- Reglas fundamentales para optimizar el uso de la caché:
  - Optimizar la localidad espacial
    - Cuanto más pequeño es el stride de acceso a un array mejor (stride óptimo = 1)

  En C las matrices se almacenan por filas. En Fortran por columnas

  - Optimizar la localidad temporal
    - Reutilizar los datos todo lo posible una vez traídos de memoria principal a memoria caché


Consideraciones caché

```
          INTEGER i,j,k,n
          REAL x(100,100,100)

          n=100
          do 30 k=2,n-1
            do 20 j=1,n
              do 10 i=1,n
                x(i,j,k)=(x(i,j,k-1)+x(i,j,k+1))/2.0
10            continue
20          continue
30        continue

          stop
          end
```

El lazo en k no es paralelo ⇒ paralelizamos el lazo j

Consideraciones caché

```
          INTEGER i,j,k,n
          REAL x(100,100,100)

          n=100
          do 30 k=2,n-1
!$OMP PARALLEL DO
            do 20 j=1,n
              do 10 i=1,n
                x(i,j,k)=(x(i,j,k-1)+x(i,j,k+1))/2.0
10            continue
20          continue
30        continue

          stop
          end
```

sincroniza n-2 veces

Consideraciones caché

Reestructuración ⇒ Solución más eficiente desde el punto de vista de paralelismo

Desde el punto de vista de acceso a memoria no recorremos las variables en el orden correcto ⇒ habría que ver qué solución es mejor

```
              INTEGER i,j,k,n
              REAL x(100,100,100)

              n=100
!$OMP PARALLEL DO
              do 20 j=1,n
                do 30 k=2,n-1
                  do 10 i=1,n
                    x(i,j,k)=(x(i,j,k-1)+x(i,j,k+1))/2.0
10              continue
30            continue
20            continue

              stop
              end
```

Consideraciones caché

- Cuando dos o más CPUs alternativa y repetidamente actualizan la misma línea caché se produce lo que se llama “contención caché” ⇒ se degrada el rendimiento del programa
- La contención cache se puede deber a dos motivos:
  - **Contención de la memoria:** 2 o más CPUs actualizan las mismas variables
  - **Falsa compartición:** las CPUs actualizan diferentes variables que se encuentran sobre la misma línea caché

La contención caché es un problema de los programas paralelos

Consideraciones caché

```
        INTEGER m,n,i,j
        REAL    a(m,n),s(m)

!$OMP PARALLEL DO PRIVATE(i,j), SHARED(s,a)
        do i=1,m
          s(i)=0.0
          do j=1,n
            s(i)=s(i)+a(i,j)
          enddo
        enddo
```

m=4

n_threads=4

El tiempo de ejecución del programa paralelo es mayor que el del secuencial debido a la "falsa compartición" de la variable s

Consideraciones caché

- Los 4 elementos de "s" están contenidos muy probablemente en la misma línea caché
- Si una CPU actualiza un elemento del array "s" modifica la línea caché que lo contiene ⇒ Todas las copias caché deben ser invalidadas ⇒ cuando otra CPU accede a esa línea debe traerla de nuevo de memoria
- El número de fallos caché aumenta de forma muy considerable

Consideraciones caché

```
INTEGER m,n,i,j
REAL    a(m,n),s(32,m)

!$OMP PARALLEL DO PRIVATE(i,j), SHARED(s,a)
do i=1,m
   s(1,i)=0.0
   do j=1,n
      s(1,i)=s(1,i)+a(i,j)
    enddo
enddo
```

Cada elemento de la variable s es ahora localizado en una línea caché diferente

Consideraciones caché

```
!$OMP PARALLEL DO PRIVATE(I),SHARED(A,B,C)
!$OMP&SCHEDULE(STATIC,1)
        DO I=1,10
           A(I)=B(I)+C(I)
        ENDDO
```

```
DO I1=1,3,5,7,9
   A(I1)=B(I1)+C(I1)
ENDDO
```

```
DO I2=2,4,6,8,10
   A(I2)=B(I2)+C(I2)
ENDDO
```

Existe compartición falsa sobre el array A

Consideraciones caché

Procesador 1:

- Escribe A(1) y marca la línea caché que lo contiene como modificada

Procesador 2:

- Detecta que la línea que contiene A(2) está modificada
- Trae la línea desde memoria
- Modifica A(2) y marca la línea como modificada

- Procesador 1:
  - Detecta que la línea que contiene A(3) está modificada
  - Trae la línea desde memoria
  - Modifica A(3) y marca la línea como modificada

Consideraciones caché

```
!$OMP PARALLEL DO PRIVATE(I),SHARED(A,B,C)
!$OMP&SCHEDULE(STATIC)
        DO I=1,10
          A(I)=B(I)+C(I)
        ENDDO
```

Una distribución por bloques mejoraría notablemente el comportamiento caché

Sobrecarga de la paralelización

- Las regiones paralelas, las directivas de reparto de trabajo y las sincronizaciones introducen sobrecarga en el código (dependiente de la implementación)
- Hay que tratar de minimizar los costes:
  - En general la directiva ATOMIC es menos costosa que la directiva CRITICAL
  - En general no será rentable paralelizar lazos con poco peso de trabajo
  - Cuando se paralelizan lazos es mejor paralelizar el lazo externo

Casos de estudio

```
          integer local_s(2,MAX_NUM_TREADS)
!$OMP PARALLEL PRIVATE(id)
          id=omp_get_thread_num()+1
!$OMP DO PRIVATE(index)
          do i=1,n
             index=MOD(ia(i),2)+1
             local_s(index,id)=local_s(index,id)+1
          enddo
!$OMP ATOMIC
          is(1)=is(1)+local_s(1,id)
!$OMP ATOMIC
          is(2)=is(2)+local_s(2,id)
!$OMP END PARALLEL
```

VERSION 1

Casos de estudio

```
          integer local_s(2)
!$OMP PARALLEL PRIVATE(local_s)
          local_s(1)=0
          local_s(2)=0
!$OMP DO PRIVATE(index)
          do i=1,n
             index=MOD(ia(i),2)+1
             local_s(index)=local_s(index)+1
          enddo
!$OMP ATOMIC
          is(1)=is(1)+local_s(1)
!$OMP ATOMIC
          is(2)=is(2)+local_s(2)
!$OMP END PARALLEL
```

VERSION 2

Casos de estudio

Tiempos de ejecución en el Superdome

	1 PE	2 PEs
versión 1	0.19s	0.37s
versión 2	0.19s	0.16s

Fallos caché en el Superdome

	1 PE		2PEs	
version 1	323570	3.75%	1498640	14%
version2	321738	3.73%	322599	1.28%

Casos de estudio

```
DO k=1,N
    DO j=1,L
        DO i=1,M
            X(i,j,k) = sqrt(X(i,j-1,k-1)) +X(i,j,k)**scale
        ENDDO
    ENDDO
ENDDO
```

- Debido a dependencias sólo el lazo más interno (lazo i) puede ser paralelizado
- El anidamiento es el adecuado para buen rendimiento serie

Casos de estudio

```
            DO k=1,N
                DO j=1,L
!$OMP PARALLEL DO PRIVATE(I), SHARED(j,k,x,scale)
                    DO i=1,M
                        X(i,j,k) =sqrt( X(i,j-1,k-1)) + X(i,j,k)**scale
                    ENDDO
                ENDDO
            ENDDO
```

VERSION 1

Sobrecarga:

- Creación y destrucción de threads ((N*L) veces)

Casos de estudio

```
!$OMP PARALLEL PRIVATE(i,j,k),SHARED(x,scale)
        DO k=1,N
          DO j=1,L
!$OMP DO
            DO i=1,M
              X(i,j,k) = sqrt(X(i,j-1,k-1)) + X(i,j,k)**scale
            ENDDO
          ENDDO
        ENDDO
!$OMP END PARALLEL
```

VERSION 2

Sobrecarga:

- Creación y destrucción de threads (sólo 1 vez)

Casos de estudio

tiempo código secuencial = 1.3 s

	4 PES	Aceleración
versión 1	0.6s	2.17
versión 2	0.4s	3.25

Análisis de eficiencia

- Herramientas para la evaluación del rendimiento:
  - Medidas de tiempo
  - gprof


Análisis de eficiencia

- gprof:
  - Válido sólo para programas secuenciales
  - Estándar en la mayoría de los sistemas UNIX
  - Muestra el número de llamadas a las diferentes funciones del programa así como sus tiempos de ejecución
  - Permite determinar qué funciones y bloques básicos del programa emplean más tiempo


Análisis de eficiencia

Pasos:

1. Compilar con la opción -pg
   gfortran –o ejecutable -pg programa.f
   gcc4 -o ejecutable –pg programa.c
2. Ejecutar el programa. Esto crea el fichero gmon.out
3. Ejecutar gprof especificando el nombre del ejecutable como argumento. Mostrará 2 tablas en la salida estándar: el *flat profile* y el *call graph*
   gprof ejecutable
4. Como las tablas pueden ser bastante largas es mejor almacenarlas en un fichero
   gprof ejecutable > gprof.out


Análisis de eficiencia

- *flat profile* (perfil plano): muestra el tiempo que el programa gasta en cada función y el número de veces que se llama a cada función
- *call graph* (grafo de llamadas): muestra para cada función qué otras funciones la llaman, a qué otras funciones llama ella, y cuántas veces sucede esto.


Análisis de eficiencia

```
program gprof
parameter(TIMES=10000)
real a,b
integer i,j

a=0.0
b=0.0

do j=1,TIMES
        a=a+1
    call calcular(a,b)
enddo
print*, 'Suma final: ',b
end
```

```
subroutine calcular(a,b)
real a,b
integer i

do i=1,10000
  call suma(a,b)
enddo
return
end

subroutine suma(a,b)
real a,b

b=a+b

return
end
```

Análisis de eficiencia

% time	cumulative seconds	self seconds	calls	self ms/call	total ms/call	name
55.64	0.34	0.34	100000000	0.00	0.00	suma_
45.67	0.62	0.28	10000	0.03	0.06	calcular_
0.00	0.62	0.00	1	0.00	617.99	MAIN__

flat profile

Análisis de eficiencia

```
index     % time    self  children    called                 name
                    0.28    0.34    10000/10000            MAIN__ [2]
[1]       100.0     0.28    0.34    10000                  calcular_ [1]
                    0.34    0.00    100000000/100000000    suma_ [4]
-----------------------------------------------
                    0.00    0.62       1/1                 main [3]
[2]       100.0     0.00    0.62       1                   MAIN__ [2]
                    0.28    0.34    10000/10000            calcular_ [1]
-----------------------------------------------
                                                           <spontaneous>
[3]       100.0     0.00    0.62                           main [3]
                    0.00    0.62       1/1                 MAIN__ [2]
-----------------------------------------------
                    0.34    0.00 100000000/100000000       calcular_ [1]
[4]        54.9     0.34    0.00 100000000                 suma_ [4]
-----------------------------------------------
```

call graph

Sesión de laboratorio

Lab 5

Más información

Benchmarking

- OpenMP microbenchmarks:
  - http://www.epcc.ed.ac.uk/research/openmpbench
- Nas Benchmarks:
  - http://www.nas.nasa.gov/Software/NPB/
- SPEC OMP2001 Benchmarks
  - http://www.spec.org/hpg/omp2001/

Otros compiladores

- Intel (http://www.intel.com/cd/software/products/asmo-na/eng/compilers/284132.htm): compilador gratuito para la comunidad universitaria (C/C++ y Fortran)
- OpenUH (http://www2.cs.uh.edu/~openuh/): compilador académico gratuito y de código abierto (C/C++ y Fortran)
- gcc (http://gcc.gnu.org/): el compilador de GNU incluye soporte para OpenMP a partir de la versión 4.2.


OpenMP 3.0

- Cambios principales:
  - Se añade la construcción **task** para permitir la paralelización a nivel de tareas y no sólo de lazos
  - Se añade la cláusula **collapse** para permitir la paralelización simultanea de varios lazos anidados
  - Mejor soporte para el anidamiento de paralelismo
    - mejor control y definición de las regiones paralelas anidadas
    - nuevas rutinas para determinar la estructura del anidamiento
  - Mejor control del tipo de planificación utilizado en la paralelización de los lazos
  - Control de threads portable a través de nuevas variables de entorno


Directiva task

- Fortran:

```
!$OMP TASK[cláusulas]
      bloque estructurado
!$OMP END TASK
```

- C/C++:

```
#pragma omp task[cláusulas]
  {
     bloque estructurado
     }
```

• Proporciona un mecanismo para definir una tarea de forma explícita

Directiva task

- Cuando un thread encuentra una directiva task genera una tarea
- La tarea puede ser ejecutada en el momento por ese thread o colocada en una cola de ejecución para ser ejecutada más tarde por cualquier thread del equipo
- Cada tarea es ejecutada por un único thread
- Se pueden anidar tareas dentro de otras tareas

Directiva task

- Cláusulas:
  - IF(expresión escalar lógica)
  - UNTIED
  - DEFAULT(PRIVATE | FIRSTPRIVATE | SHARED | NONE)
  - PRIVATE(lista de variables)
  - FIRSTPRIVATE(lista de variables)
  - SHARED(lista de variables)

Directiva task

```
#define LARGE_NUMBER 10000000
double item[LARGE_NUMBER];

int main() {
#pragma omp parallel
{
    #pragma omp single
    {
            int i;
            for (i=0; i<LARGE_NUMBER; i++)
                    #pragma omp task // i is firstprivate, item is shared
                            process(item[i]);
    }
}
}
```

Directiva task

Directiva **TASKWAIT**

- Fortran:

  !$OMP TASKWAIT

- C/C++:

  #pragma omp taskwait

- Espera hasta la compleción de todas las tareas generadas desde el inicio de la tarea actual
- La barrera (**barrier**) implícita o explícita incluye un **taskwait** implícito


Cláusula collapse

- Permite la paralelización simultánea de lazos perfectamente anidados (loop collapse)
- **Ejemplo:**

```
#pragma omp parallel for private (i,j)
        for (i=0;i<N;i++){
                for(j=0;j<N;j++){
                        !cuerpo del lazo, usa i y j
                }
        }
```


Cláusula collapse

Paralelización simultánea usando OpenMP 2

```
#pragma omp parallel for private (ij,i,j)
        for (ij=0;ij<N*N;ij++){
                i=ij/N;
                j=ij%N;
                !cuerpo del lazo, usa i y j
        }
```

Cláusula collapse

Paralelización con OpenMP 3

```
#pragma omp parallel for private (i,j) collapse(2)
        for (i=0;i<N;i++){
                for(j=0;j<N;j++){
                        !cuerpo del lazo, usa i y j
                }
        }
```

Nuevo soporte para el anidamiento de paralelismo

- Ahora omp_set_num_threads(), omp_set_dynamic() y omp_set_nested() puede ser llamadas desde dentro de una región paralela
  - Afectan al siguiente nivel de paralelismo
- Se añaden variables de entorno y rutinas runtime para establecer y conocer el máximo número de niveles de paralelismo anidados activo
  - OMP_MAX_ACTIVE_LEVELS
  - omp_set_max_active_levels()
  - omp_get_max_active_levels()


Extensión de las formas de planificar un lazo

- En la versión 3.0 la distribución de las iteraciones de un lazo con una cláusula *schedule static* es determinista
- Esto garantiza que el siguiente trozo de código funciona

```
!$ompdoschedule(static)
do i=1,n
a(i) = ....
end do
!$ompend do nowait
!$omp do schedule(static)
do i=1,n
.... = a(i)
end do
```


Extensión de las formas de planificar un lazo

- Se añade un nuevo tipo de planificación (schedule **auto**) que da total libertad al runtime para determinar la distribución de las iteraciones entre los threads
- Se añaden las rutinas omp_set schedule() y omp_get_schedule()

Control portable de *threads*

- Control del tamaño del stack

  `export OMP_STACK_SIZE 64M`
- Control del comportamiento de los threads mientras esperando

  `export OMP_WAIT_POLICY ACTIVE`
  - Opciones: ACTIVE, PASSIVE
- Control del máximo número de threads participando en el programa

  export OMP_THREAD_LIMIT

  omp_get_thread_limit()

Referencias

- Página oficial OpenMP:
  - http://www.openmp.org
- La comunidad de OpenMP:
  - http://www.compunity.org
- Manuales HP:
  - http://docs.hp.com
- Libro OpenMP:
  - Parallel Programming in OpenMP. Rohit Chandra et al. Morgan Kaufmann, 2000.
  - Parallel Programming in C with MPI and OpenMP. Michael J.Quinn. McGraw-Hill, 2003.

PROGRAMACIÓN OPENMP

EJERCICIOS DE LABORATORIO

Introducción

Los siguientes ejercicios forman las prácticas guiadas del curso PROGRAMACIÓN SOBRE SISTEMAS DE MEMORIA COMPARTIDA: PROGRAMACIÓN CON OPENMP

No se proporcionará ningún documento con las soluciones a estos ejercicios. Asegúrese de que los comprende y sabe cómo resolverlos, y no dude en consultar al profesor en caso de duda.

Instalación de los ejercicios

Los ejercicios de laboratorio están localizados en el directorio /tmp de la máquina *svgd*, en un fichero empaquetado llamado laboratorios_svgd.tar. Copie el fichero a su cuenta y su directorio local y desempaquételo de la siguiente forma:

```
>tar –xvf laboratorios_svgd.tar
```

Ahora debería tener un directorio llamado laboratorios_svgd en su directorio local. Dentro encontrará los ejercicios clasificados en carpetas, según el tema al que hagan referencia (lab1, lab2,...).

LAB 1. Toma de contacto con el computador SVGD

- Compilación

    1. En el directorio lab1 encontrarás 2 programas secuenciales, uno en C (area.c) y otro en Fortran (matriz.f). Compílalos y genera dos ejecutables llamados *area* y *matriz* respectivamente.

- Ejecución en interactivo

    2. El programa **area** calcula el área de un triángulo rectángulo. Para ello solicita como argumentos la base y la altura del triángulo e imprime el resultado. Ejecútalo en interactivo para ver como funciona.
    3. El programa **matriz** lee una matriz tridimensional almacenada en el fichero matriz3d y actualiza sus valores. Los nuevos valores los almacena en el fichero **salida**. Ejecútalo en interactivo para ver como funciona.

- Ejecución utilizando las colas

    4. Envía el programa **area** a colas especificando que se almacene la salida en el fichero **area.txt.**
    5. Envía el programa **matriz** a colas teniendo en cuenta que debe leer los datos de entrada del fichero **matriz3d** almacenado en $HOME/laboratorios_svgd/lab1.
    6. Comprueba el estado de los trabajos utilizando *qstat*.
    7. Envía de nuevo el programa **matriz** a colas pero esta vez para medir su tiempo de ejecución.

LAB 2. Paralelización de programas utilizando directivas de reparto de trabajo

- Creación de regiones paralelas

8. Revisa el código del programa omp_hello.c y omp_hello.f fijándote en las directivas y las llamadas a las rutinas de OpenMP.
9. Compila uno de los códigos y ejecútalo sobre diferente número de threads.

- Entorno de datos

10. Compila el programa prac1.f y ejecútalo sobre diferente número de threads ¿Cuál es el problema? Corrígelo y ejecútalo de nuevo (editor disponible: GNU nano).
11. Compila el programa prac2.f, ejecútalo varias veces y compara los resultados de salida para diferente número de threads ¿Cuál es el problema? Corrígelo y ejecútalo de nuevo.

- Paralelización de lazos

12. Indica cual o cuales de los siguientes lazos no son adecuados para ejecución paralela y por qué:

a)

```
for (i=0; i< (int) sqrt(x); i++){
        a[i]=2.3*i;
        if (i<10) b[i] = a[i];
}
```

b)

```
flag=0;
for (i=0; (i<n) & (!flag); i++) {
        a[i]=2.3*i;
        if (a[i] < b[i]) flag=1;
}
```

c)

```
for (i=0; i<n; i++)
        a[i]=foo(i);
```

d)

```
for (i=0; i<n; i++) {
        a[i]=foo(i);
        if (a[i] < b[i]) break;
```

e)

```
for (i=0; i<n; i++) {
        a[i]=foo(i);
        if (a[i] < b[i]) a[i]=b[i];
```

f)

```
dotp=0;
for(i=0; i<n; i++)
        dotp+=a[i]*b[i];
```

g)

```
for(i=k; i<2*k; i++)
        a[i]=a[i]+a[i-k];
```

h)

```
for(i=k; i<n; i++)
        a[i]=b*a[i-k];
```

13. El programa mult.c realiza la multiplicación de 2 matrices (D=AxB). Paralelízalo y mide la aceleración obtenida sobre diferente número de threads.
14. Paraleliza el código del programa mandel.c ¿Qué aceleración obtienes?
15. El programa pi.f utiliza integración numérica para estimar el valor de Π. Paraleliza el código y calcula la aceleración obtenida.

LAB 3. Paralelización de programas utilizando directivas de reparto de trabajo (continuación)

- Paralelización de lazos

16. El programa planificacion.f ya tiene incluidas las directivas de paralelización. Compílalo y ejecútalo para 1, 2, 3 y 4 procesadores. Compara los tiempos de ejecución ¿Qué está ocurriendo?
17. Prueba en el programa anterior diferentes políticas de planificación para la paralelización del segundo lazo. Compara los tiempos de ejecución ¿Cuál es la que mejor funciona? ¿Por qué?
18. Paraleliza el siguiente segmento de código utilizando pragmas:

```
for (i=0; i<m; i++) {
        rowterm[i]=0.0;
        for (j=0;  j <p; j++)
                rowterm[i]+=a[i][2*j]*a[i][2*j+1];
}
for(i=0; i<q; i++){
        colterm[i]=0.0;
        for(j=0; j<p; j++)
                colterm[i]+=b[2*j][i]*b[2*j+1][i];
}
```

- Secciones paralelas

19. El programa mxm.c realiza 2 multiplicaciones de matrices (D=AxB y E=AxC). Paralelízalo utilizando secciones paralelas de forma que cada una de las multiplicaciones se realice en una sección y almacena el código paralelo como mxm_omp.c. Compila y ejecuta sobre diferente número de threads ¿Consigues aceleración?

LAB 4. Directivas de sincronización, funciones de la librería y variables de entorno

- Directiva THREADPRIVATE

    20. ¿Qué valor crees que se imprimirá al ejecutar el programa threadprivate.f?. ¿Y si se cambia la directiva MASTER por una directiva SINGLE? Compruébalo (ejecuta varias veces utilizando 4 threads).

- Rutinas OpenMP

    21. Copia el programa paralelo mxm_omp.c generado en el LAB 2 a este directorio y modifícalo para que imprima qué número de thread ejecuta cada sección.
    22. El programa jacobi.f consta básicamente de tres subrutinas (initialize, jacobi y error_check). Mide el tiempo de ejeucución consumido por cada una de las subrutinas utilizando las rutinas de medida de tiempos que facilita OpenMP.
    23. Paraleliza la subrutina jacobi del programa jacobi.f y mide tiempos de ejecución para diferente número de threads de dicha subrutina.

- Directivas de sincronización

    24. Propón dos formas correctas de paralelizar el código reduction.f (una utilizando la cláusula REDUCTION y otra utilizando secciones críticas) ¿Se te ocurren más alternativas?
    25. Compila y ejecuta el programa barrier.f ¿Es correcta la salida? ¿Por qué?
    26. Reescríbe el programa critical.f utilizando la directiva *atomic* en vez de *critical*. Reescríbelo utilizando *locks* en vez de secciones críticas. Evalúa el rendimiento de las tres versiones con diferente número de threads.

- Variables de entorno OpenMP

    27. ¿Proporciona el SVGD ajuste dinámico del número de threads? Crea un programa en C o Fortran que imprima el número máximo de threads sobre el que se ejecuta una región paralela, el número de procesadores disponibles, y el número de threads sobre el que realmente se ejecuta la región paralela. Ejecútalo varias veces con el ajuste dinámico activado (utiliza la variable de entorno OMP_DYNAMIC) modificando el número de threads de ejecución (utiliza la variable de entorno OMP_NUM_THREADS) ¿Qué observas?

LAB 5. Paralelización a nivel de lazo

26. Prueba gprof con el programa de ejemplo simple.f para reproducir los resultados vistos en teoría.
27. El programa md.f implementa la simulación de un proceso simple de dinámica molecular. Consta de más de 200 líneas de código y está compuesto de varias subrutinas y funciones. La paralelización en este tipo de códigos debe llevarse a cabo de forma incremental, empezando por abordar la paralelización de los lazos más costosos. Para ello conviene utilizar herramientas como gprof que nos informen sobre la estructura del código y su peso computacional. En este ejercicio se trata de que utilices la información proporcionada por gprof ¿Cuál es la subrutina que consume la mayor parte del tiempo de ejecución? ¿Cuántas veces se llama? ¿Tiene dentro llamadas a otras subrutinas?.
28. Paraleliza el código del programa md.f y mide el rendimiento que se obtiene.