OpenMP

Cálculo Científico con computadoras paralelas

Victorio E. Sonzogni

CIMEC Centro Internacional de Métodos Computacionales en Ingeniería
CONICET-UNL, FICH, Santa Fe, Argentina

OpenMp

- OpenMP (Open Multi- Processing)
- Es una *API* (Application Programming Interface)
- Permite, mediante *directivas al compilador*, agregadas al programa, procesarlo en paralelo a través de *hilos* de ejecución en computadoras de memoria compartida.

OpenMp

- OpenMP está soportado por un conjunto de empresas:
  - De hardware:
    Intel, HP, IBM, SUN, SGI, Compaq
  - De software:
    PGI, KAI, GCC, PSR, APR, Absoft
  - De programas de aplicación:
    ANSYS, NAG, DOE, ASCI, etc.

OpenMP

- OpenMP posee un conjunto de
  - Directivas al compilador
  - Librerías de funciones y subrutinas
  - Variables de entorno
- Que pueden ser usadas desde el programa de aplicación escrito en:
  - Fortran
    Fortran 77, Fortran 90, Fortran 95
  - C
    C90, C99
  - C++

OpenMP Directivas

Las directivas al compilador de OpenMP tiene la siguiente sintaxis, donde *construct* representa una directiva:

- En C/C++
  #pragma omp *construct [clause [, clause] ...]*
- En Fortran
  c$omp *construct [clause [, clause] ...]*
  !$omp *construct [clause [, clause] ...]*
  ***$omp** *construct [clause [, clause] ...]*
- El programa puede ser compilado por un compilador estándar, que simplemente ignora las directivas (comentarios)

Directivas de OpenMP

Las directivas de OpenMP pueden clasificarse en 5 categorías:

- Creación de regiones paralelas
- Ejecución de trabajo compartido
- Clases de datos
- Sincronización
- Funciones en tiempo de ejecución y variables de entorno

El mismo nombre de las directivas se usa en Fortran y en C/C++

Creación de regiones paralelas

Directiva **parallel**

- Sintaxis en C/C++

  #pragma omp parallel *[clause [, clause] ...] newline*
  bloque estructurado

- Sintaxis en Fortran

  !$omp parallel *[clause [, clause] ...]*
  bloque structurado
  !$omp end parallel

Creación de regiones paralelas

Bloque estructurado

- Es un bloque de sentencias que tiene una sola entrada y una sola salida.
- En un bloque estructurado puede haber una llamada a **exit()** (en C/C++) o un **stop** en Fortran
- Pero no puede haber ramificaciones que saquen la ejecución del bloque (o la introduzcan en él).

Directiva parallel

La directiva *parallel* puede estar acompañada por las siguientes cláusulas:

- **if** (*expresion lógica escalar*)
- **num_threads** (*expresion entera*)
- **private** (*lista*)
- **firstprivate** (*lista*)
- **share** (*lista*)
- **default** (**shared** | **none**)
- **copyin** (*lista*)
- **reduction** (*oper:lista*)

Directiva parallel

- La cantidad de hilos de ejecución que se abren con la directiva *parallel* puede estar dado en la cláusula **num_threads** Antes de esa directiva se ejecuta un solo hilo del programa, y al salir de la region paralela continua también un solo hilo.
- La lista de variables *compartidas* por todos los hilos está indicada en **share** y la de variables privadas a cada hilo, en **private**
- Las variables de reducción se indican con **reduction** , junto al tipo de operación

Ejemplo: Hola mundo

```
      program hello
      implicit none
      integer  ih, nh
      integer  omp_get_thread_num
      integer  omp_get_num_threads
      include "omp_lib.h"
!
!$omp parallel private (ih,nh) num_threads (4)
      ih = omp_get_thread_num ()
      nh = omp_get_num_threads ()
      write (*,*)'Soy el hilo = ', ih, ' de ', nh
!$omp end parallel
      end program
```


Ejemplo: Hola mundo

```
#include <iostream>
#include <omp.h>
main ( ) {
  int  ih, nh ;
  cout << endl ;
#pragma omp parallel private (ih, nh) num_threads (4)
  {
    ih = omp_get_thread_num ();
    nh = omp_get_num_threads ();
    cout << "soy el hilo = " << ih << endl ;
    cout << "de = " << nh << endl ;
  } // end pragma: todos los hilos se unen al master y fin
} // end main
```

Ejemplo: Hola mundo

- La directiva

  ```
  !$omp parallel private (ih,nh)
  ```

  abre 4 hilos de ejecución paralelos.
- Las variables `ih, nh` son privadas a cada hilo (una copia diferente en memoria para cada uno).
- La función

  ```
  ih = omp_get_thread_num ()
  ```

  devuelve el número de orden de este hilo.
- La función

  ```
  nh = omp_get_num_threads ()
  ```

  devuelve el número total de hilos paralelos.

Ejecución de trabajo compartido

- Ejecución paralela de ciclos
  for (C/C++) o *do* (Fortran)
- Secciones paralelas
  section
- Directiva *single*
- Directiva *master*
- Directiva *workshare*

Estas directivas deben ser llamadas dentro de una región paralela.

Ejecución de ciclos en paralelo

- Sintaxis en C/C++

  #pragma omp for *[clause [, clause] ...]*
  cuerpo del ciclo

- Sintaxis en Fortran

  !$omp do *[clause [, clause] ...]*
  cuerpo del ciclo
  !$omp end do [nowait]

Estas directivas deben aparecer antes de la sentencia *for* (en C/C++), o *do* (en Fortran), para que la tarea de esos ciclos se reparta entre los procesos paralelos.

Ejecución de ciclos en paralelo

Las cláusulas que acompañan al *omp for* o *omp do* pueden ser:

- **private** (*lista*)
- **firstprivate** (*lista*)
- **lastprivate** (*lista*)
- **reduction** (*oper:lista*)
- **ordered**
- **schedule** (*kind[, chunk]*)
- **nowait**

Ejecución de ciclos en paralelo

- Las tres primeras definen las variables privadas y si éstas conservan su valor al entrar o al salir del ciclo.
- La clausula *reduction* indica las variables de reducción y la operación asociada.
- La clausula *ordered* debe aparecer is hay una zona *ordenada* dentro del ciclo
- La clausula *schedule* indica cómo se distribuye el trabajo y puede definirse los tipos: *static* , *dynamic* , *guided* y *runtime* .
- La clausula *nowait* elimina una barrera implícita que existe al finalizar el ciclo.

Distribución de índices del ciclos

Las formas de diagramado:

- **Schedule (static)**
  Divide el total de iteraciones del ciclo entre los procesos paralelos, asignando grupos aproximadamente iguales de índices contiguos a cada proceso.
- **Schedule (static, chunk)**
  Divide el total de iteraciones del ciclo en tamaños dados por *chunk* y los va entregando a cada proceso en forma ordenada, al inicio del ciclo (estáticamente).
- **Schedule (dynamic, chunk)**
  Divide el total de iteraciones del ciclo en tamaños dados por *chunk* y los va entregando uno a cada proceso. A medida que cada proceso termina su tarea, les va entregando nuevos grupos de índices. Por default *chunk*$= 1$.
- **Schedule (guided, chunk)**
  Similar al *dynamic*, pero asignando grupos de tamaño mayor al principio y menores luego, hasta terminar con grupos de tamaño *chunk*.
- **Schedule (runtime)**
  El diagramado se define al momento de ejecución. Lo toma de una variable de control.

Secciones paralelas

- Sintaxis en C/C++

#pragma omp sections *[clause [, clause] ...]*
{
*[***#pragma omp section** *]*
bloque estructurado
*[***#pragma omp section** *]*
bloque estructurado
. . .
}

Secciones paralelas

- Sintaxis en Fortran

!$omp sections *[clause [, clause] ...]*
*[***!$omp section** *]*
bloque estructurado
*[***!$omp section** *]*
bloque estructurado
. . .
!$omp end sections [nowait]

Secciones paralelas

- Cada sección es ejecutada en un hilo en paralelo
- Las cláusulas de las secciones paralelas pueden ser:
  - **private** (*lista*)
  - **firstprivate** (*lista*)
  - **lastprivate** (*lista*)
  - **reduction** (*oper:lista*)
  - **nowait**

Directiva *single*

- Sintaxis en C/C++

  #pragma omp single *[clause [, clause] ...]*
  bloque estructurado

- Sintaxis en Fortran

  !$omp single *[clause [, clause] ...]*
  bloque estructurado
  !$omp end single [nowait]

Directiva *single*

- La sección definida por *single* es ejecutada por un solo hilo. (no necesariamente el master).
- Las cláusulas de las sección *single* pueden ser:
  - **private** (*lista*)
  - **firstprivate** (*lista*)
  - **copyprivate** (*lista*)
  - **nowait**
- La directiva *master* es similar a *single*, pero ordena que sea ejecutada por el hilo maestro.

Directiva *workshare*

- Sintaxis en Fortran

!$omp workshare
bloque estructurado
!$omp end workshare [nowait]

- Esta construcción divide la ejecución del bloque estructurado en unidades de trabajo que asigna a cada hilo en paralelo.

Directivas combinadas

Hay directivas que combinan *parallel* y *for* o *do*

- Sintaxis en C/C++

  #pragma omp parallel for *[clause [, clause] ...]*
  cuerpo del ciclo

- Sintaxis en Fortran

  !$omp parallel do *[clause [, clause] ...]*
  cuerpo del ciclo
  !$omp end parallel do [nowait]

Estas directivas equivalen a crear una zona paralela y luego ejecutar un ciclo en paralelo. (combina *parallel* y *for* o *do*)
Igualmente se puede combinar *parallel* y *sections*

Clases de datos

- Hay variables *compartidas* a la cual tienen acceso todos los hilos de ejecución del programa
- Hay variables *privadas* para la cual cada hilo tiene una copia de esa variable en la memoria.
- La mayoría de las variables son *compartidas* por default
- Las variables globales (*Common* de Fortran o variables estáticas de C) son compartidas.
- Las variables definidas dentro de subrutinas, dentro de una zona paralela, son *privadas*.
- Los índices de los ciclos (*for* o *do*) son privados.

Clases de datos

Se puede cambiar el tipo de datos usando clausulas de cada directiva.

- La clausula *shared*, en regiones paralelas, define variables compartidas.
- La clausula *private*, define variables privadas.
- La clausula *firstprivate*, define variables privadas y les asigna el valor que tenia esa variable antes de invocar a la directiva que posee esta clausula.
- La clausula *lastprivate*, define variables privadas y cuando sale de la zona paralela, queda como global con el último valor alcanzado.
- El status por default puede ser modificado con la clausula *default(private* | *shared* | *none)*.

Variable privada

- La clausula *private* crea una copia local de la variable en cada hilo.
- El valor no es inicializado.

```
      k = 0
!$omp parallel do private (k)
      do i=1,100
      k = k + i
      end do
!$omp end parallel do
      write (*,*) k
```

- La sentencia `k = k + i` es incorrecta pues la variable k no está inicializada.
- La sentencia `write (*,*) k` es incorrecta pues la variable k no tiene valor.

Variable *firstprivate*

- La clausula *firstprivate* crea variable privada, pero le asigna el valor que tenía en el hilo maestro.

```
      k = 0
!$omp parallel do firstprivate (k)
      do i=1,100
      k = k + i
      end do
!$omp end parallel do
      write (*,*) k
```

- La sentencia `k = k + i` es correcta ahora. Cada hilo tiene su copia de la variable k, con valor inicial 0.
- La sentencia `write (*,*) k` es incorrecta pues la variable k no tiene valor.

Variable *lastprivate*

- La clausula *lastprivate* crea variable privada, y pasa el valor que toma en la última iteración a una variable global.

```
      k = 0
!$omp parallel do firstprivate (k) lastprivate (k)
      do i=1,100
      k = k + i
      end do
!$omp end parallel do
      write (*,*) k
```

- La sentencia `k = k + i` es correcta ahora. Cada hilo tiene su copia de la variable k, con valor inicial 0.
- La sentencia `write (*,*) k` es correcta pues la variable k ahora está definida, con el último valor (ej: 100).

Variables privadas y compartidas

```
      a = 1
      b = 1
      c = 1
      d = 1
!$omp parallel private (b) firstprivate (d) shared (c)
      ...
!$omp end parallel
```

- Dentro de la zona paralela:
  - Las variables a y c son compartidas y tienen valor 1.
  - La clausula *shared (c)* es redundante
  - La variable b es privada y con valor indefinido
  - La variable d es privada y tiene valor 1.
- Fuera de la zona paralela:
  - Las variables b y d son indefinidas

Sincronización

- Hay varias directivas que permiten sincronizar los hilos de ejecución:
  - atomic
  - barrier
  - critical
  - flush
  - ordered

Sincronización

- Directiva *critical*
  - Todos los hilos ejecutan la zona crítica, pero uno a la vez. (No puede haber simultaneamente más de un hilo)

```
      sum = 0
!$omp parallel private (ih) shared (sum)
      ih = omp_get_thread_num ()
!$omp critical
      sum = sum + ih
!$omp end critical
!$omp end parallel
```


Sincronización

- Directiva *barrier*
  - Al llegar a una barrera, los hilos deben esperar hasta que todos los demás hayan llegado allí, antes de continuar.
  - Sintaxis.

```
!\$omp barrier

#pragma omp barrier
```

  - Al final de las regiones paralelas o de ejecución compartida (ciclos paralelos) hay barreras implícitas.
  - Si se quiere levantar esa barrera implícita se puede usar la clausula *nowait*

Sincronización

- Directiva *order*: en un ciclo en paralelo, especifica que esa region debe ser ejecutada en el orden de las iteraciones del ciclo.
- Directiva *atomic*: especifica que una variable en memoria debe ser actualizada atomicamente, evitando escritura simultánea.
- Directiva *flush*: actualiza la vista que cada hilo tiene del contenido de la memoria.

Funciones en tiempo de ejecución

- *omp_set_num_threads*: define la cantidad de hilos de ejecución a ser usados.
- *omp_get_num_threads*: pregunta cuántos hilos de ejecución hay.
- *omp_get_thread_num*: pregunta cuál es este hilo entre todos.
- *omp_get_num_procs*: pregunta cuántos procesadores hay.
- *omp_in_parallel*: pregunta si está dentro de una zona paralela.
- *omp_set_dynamic*: habilita/deshabilita el ajuste dinámico de la cantidad de hilos.

Funciones en tiempo de ejecución

- *omp_init_lock* y *omp_destroy_lock*: inicializa una variable compartida par ser usada como traba.
- *omp_set_lock* y *omp_unset_lock*: traba o destraba la variable compartida, creando así una zona protegida.
- *omp_get_wtime*: retorna el tiempo transcurrido (*reloj de pared*), con respecto a un tiempo inicial arbitrario.
- *omp_get_wtick*: retorna la precisión del reloj usado en *omp_get_wtime*

Variables de entorno

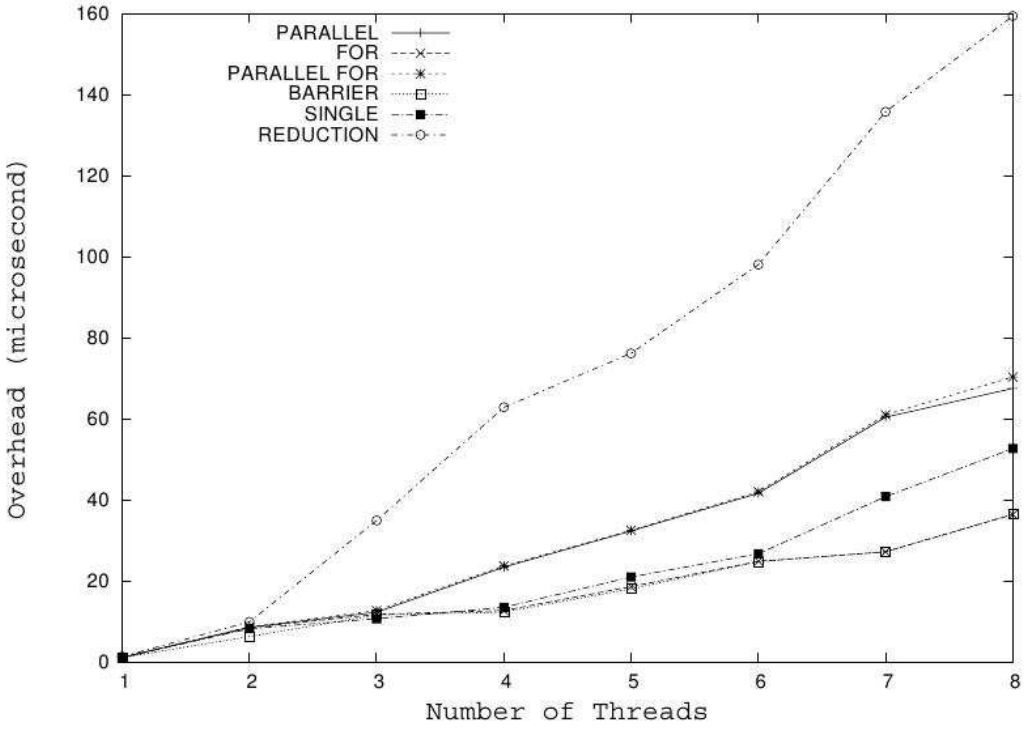
- *OMP_SCHEDULE*: contiene una variable d eocntrol para el diagramado en paralelo (*schedule type* y *chunk*)
- *OMP_NUM_THREADS*: contiene una variable con la cantidad de hilos a usar en regiones paralelas.
- *OMP_DYNAMIC*: define variable de control para elajuste dinámico de la cantidad de hilos.
- *OMP_NESTED*: define variable de control para habilitar o deshabilitar paralelismo imbricado.

Factores que influyen en el rendimiento

- Forma de acceso a la memoria
  Uso del cache. Re-uso de datos
- Fracción secuencial o redundante del programa
- Sobrecarga de paralelización
  Tiempo para manejar las construcciones de OMP (creación/destrucción de hilos de ejecución; distribución de datos; etc.)
- Desequilibrio entre puntos de sincronización.
- Sobrecarga por sincronización (tiempos a la espera de acceso a zonas críticas, etc.)

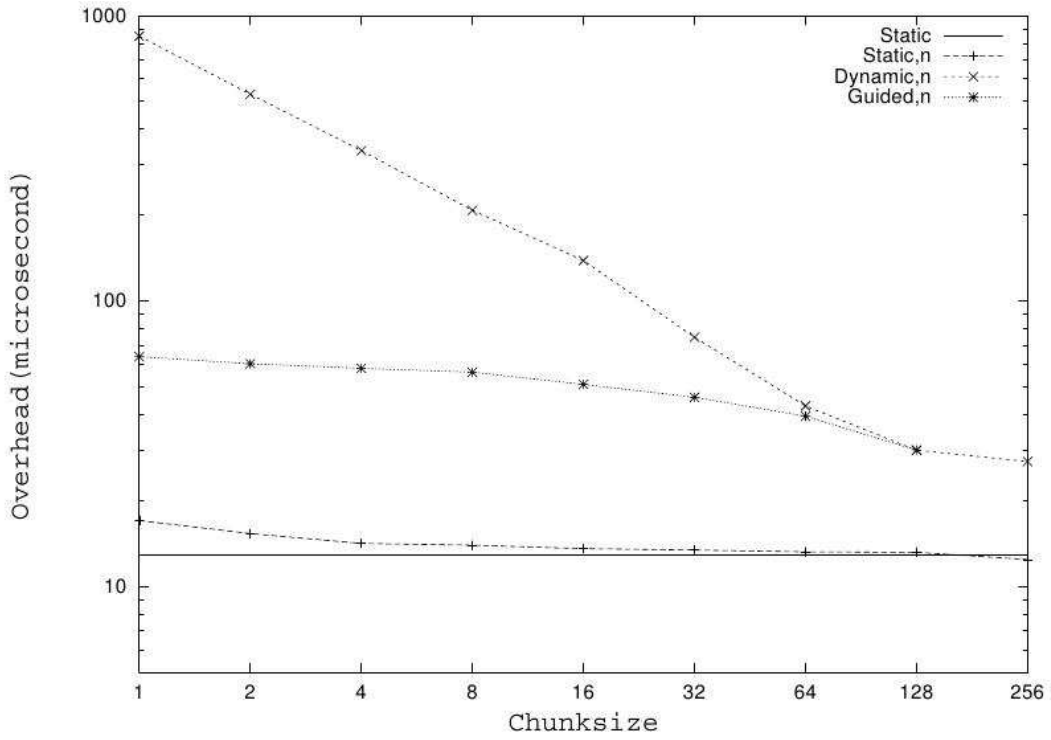
Sobrecarga de construcciones OMP

- Hay benchmarks para estimar los tiempos de sobrecarga: EPCC, SPHINX, etc.



Sobrecarga de construcciones OMP

- Sobrecarga por la forma de distribución de datos



Uso eficiente de OMP

- La construcción *barrier* es cara. Si se puede eliminar, mejor. La clausula *nowait* permite eliminar el *barrier* implícito en varias construcciones OMP.
- Si se puede evitar la clausula *order* se ahorra tiempo de sobrecarga. Por ejemplo: se puede intentar escribir los resultados fuera de una zona paralela.
- Las zonas críticas largas aumentan la probabilidad de que los hilos de ejecución tengan que esperar. Conviene sacar operaciones que no precisen estar en zonas críticas.
- Maximizar las regiones paralelas. Si hay varios ciclos uno despues de otro, no iniciar una zona paralela en cada uno, sino ponerlos todos dentro de una única zona paralela.
- Tratar de poner regiones paralelas en los ciclos más externos.

Uso eficiente de OMP

Ejemplos:

- En este caso, se elimina la barrera en el segundo ciclo :

```
#pragma omp parallel
{
        .........
   #pragma omp for
   for (i=0; i<n; i++)
        .........
   #pragma omp for nowait
   for (i=0; i<n; i++)

} /*-- End of parallel region - barrier is implied --*/
```


Uso eficiente de OMP

- En este caso, se reduce el número de barreras :

```
#pragma omp parallel default(none) \
        shared(n,a,b,c,d,sum) private(i)
{
   #pragma omp for nowait
   for (i=0; i<n; i++)
       a[i] += b[i];
   #pragma omp for nowait
   for (i=0; i<n; i++)
       c[i] += d[i];
   #pragma omp barrier
   #pragma omp for nowait reduction(+:sum)
   for (i=0; i<n; i++)
    sum += a[i] + c[i];
} /*-- End of parallel region --*/
```

Uso eficiente de OMP

- En este caso, la instrucción `c = d * d` involucra sólo variables privadas y puede sacarse de la sección crítica.

```
#pragma omp parallel shared(a,b) private(c,d)
{
      ......
#pragma omp critical
    {
      a += 2 * c;
      c = d * d;
    }
} /*-- End of parallel region --*/
```


Uso eficiente de OMP

- En este caso, el programa abajo es preferible frente al de la página siguiente ya que este último tiene mayor sobrecarga por `fork-join` y barreras implícitas.

```
#pragma omp parallel
{
    #pragma omp for /*-- Work-sharing loop 1 --*/
    { ...... }

    #pragma omp for /*-- Work-sharing loop 2 --*/
    { ...... }

       .........

    #pragma omp for /*-- Work-sharing loop N --*/
    { ...... }
}
```

Uso eficiente de OMP

```
#pragma omp parallel for
for (.....)
{
     /*-- Work-sharing loop 1 --*/
}
#pragma omp parallel for
for (.....)
{
     /*-- Work-sharing loop 2 --*/
}
     .........
#pragma omp parallel for
for (.....)
{
     /*-- Work-sharing loop N --*/
}
```


Uso eficiente de OMP

- En este caso, el overhead (por `fork-join`) se realiza n^2 veces.

```
for (i=0; i<n; i++)
    for (j=0; j<n; j++)
        #pragma omp parallel for
        for (k=0; k<n; k++)
            { .........}
```

Conviene esta otra forma:

```
#pragma omp parallel
    for (i=0; i<n; i++)
        for (j=0; j<n; j++)
            #pragma omp for
            for (k=0; k<n; k++)
                { .........}
```

Ejemplo: Producto matriz vector

En C:

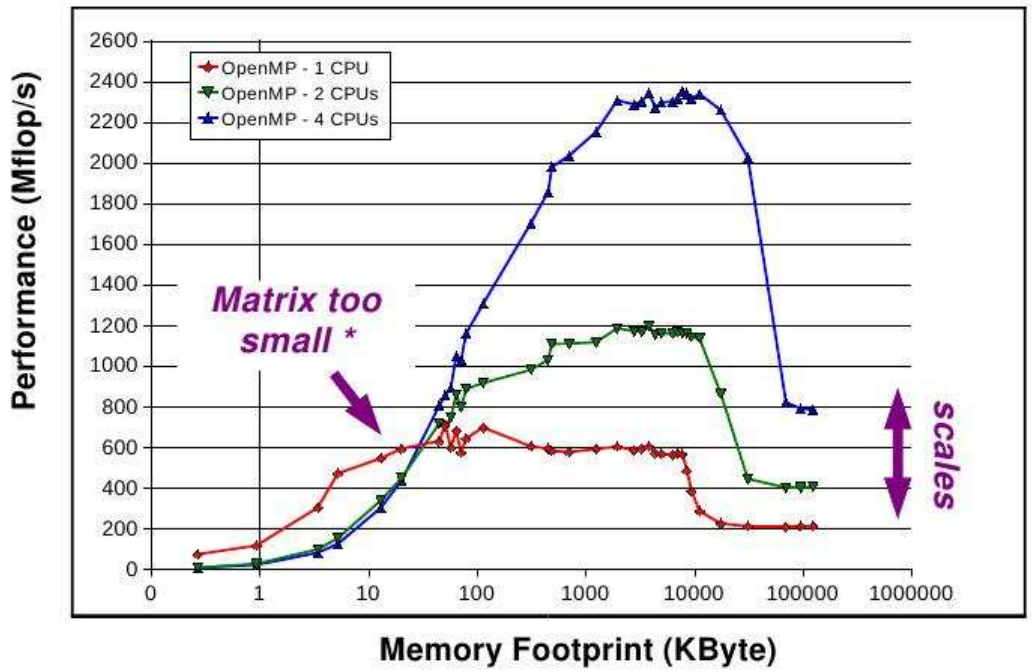
```
void mxv(int m, int n, double * restrict a,
         double * restrict b, double * restrict c)
{
   int i, j;
#pragma omp parallel for default(none) \
        shared(m,n,a,b) private(i,j)
   for (i=0; i<m; i++)
   {
    a[1] = b{i*n]*c(0);
    for (j=1; j<n; j++)
      a[i] += b[i*n+j]*c(j);
   }
}
```


Ejemplo: Producto matriz vector

En Fortran:

```
      subroutine mxv(m, n, a, b, c)
      implicit none
      integer m, n, i, j
      real a(m), b(m,n), c(n)
!$OMP PARALLEL DO DEFAULT(NONE) &
!$OMP SHARED(m,n,a,b,c) PRIVATE(i,j)
      do i = 1, m
        a(i) = b(i,1) * c(1)
        do j = 2,n
          a(i) = a(i) + b(i,j) * c (j)
        end do
      end do
!$OMP END PARALLEL DO
      return
      end
```

Ejemplos



*) With the IF-clause in OpenMP this performance degradation can be avoided

Ejemplos

- En la figura (tomada de B. Chapman, G. Jost & R. Van Der Pas) se puede ver el desempeño del programa en C para diferente cantidad de procesadores.
- Por debajo de 50Kbytes de memoria requerida (aprox. matrices de 80*80), la sobrecarga del paralelismo hace que no sea conveniente el procesado en paralelo.
- Esto puede prevenirse en un código paralelo con una clausula IF que solo ejecute en paralelo si el tamaño de la matriz es superior a ese umbral.
- Para tamaños de matrices grandes, escala linealmente.
- Hay un rango intermedio en que el escalado es superlineal.
- Esto se debe a que al agregar procesadores aumenta el espacio en cache de memoria. Pero superado ese límite el escalado pasa a ser lineal.