Tema 4.2 Programación mediante directivas OpenMP: Paralelismo en Datos

Computación de Altas Prestaciones

Carlos García Sánchez

5 de octubre de 2022

- 'Using OpenMP : portable shared memory parallel programming", Barbara Chapman, et all. 2008
- "OpenMP 5.0", `https://www.openmp.org/wp-content/uploads/OpenMP-API-Specification-5.0.pdf`

Outline

Formato

`#pragma omp construct [clauses]`

- Una directiva válida (`construct`) debe de aparecer después del *pragma*
- Las clausulas son opcionales
- Se permite una nueva línea para continuar la construcción añadiendo \ al final de la línea predecente
- La construcción suele estar delimitada por llaves para indicar el bloque

Directiva `parallel`

- `#pragma omp parallel [clauses]`

Clausulas

- *num_threads (integer-expression)*
  - Número de hilos fijado, en su defecto definido previamente con `omp_set_num_threads()` o variable entorno `OMP_NUM_THREADS`
- *if(scalar-expression)*
- *shared (var-list)*
- *firstprivate (var-list)*
- *default(none|shared|private|firstprivate)*
- *reduction(var-list)*

Compartición de datos

- Corresponde al número de clausulas en la construcción paralela
  - *shared*
  - *private*
  - *firstprivate*
  - *default*
  - *threadprivate*
  - *lastprivate*
  - *reduction*

Compartición de datos (*shared*)

shared

- Cuando una variable está definida como **shared** dentro de la región paralela es la misma que fuera de ella
  - En la región paralela: *todos los hilos ven la misma variable*
    - NOTA: no necesariamente el mismo valor
  - Normalmente se necesita alguna clase de sincronización para actualizarla

Compartición de datos (*shared*)

¡¡¡ Imprime 2 o 3 !!!

example_shared.c

```
int x=1;
#pragma omp parallel shared(x) num_threads(2)
{
   x++;
   printf("%d\n" , x);
}
printf("%d\n" , x);
```

Compartición de datos (*private*)

- Cuando una variable está definida como **private**, la variable se replica en la región paralela con una nueva variable del mismo tipo con un valor indefinido
  - Esto significa que todos los hilos tienen una variable diferente
  - Se puede acceder sin ningún tipo de sincronización

Compartición de datos (private)

Puede imprimir cualquier cosa (¿?, 2 o 3)

example_private.c

```
int x=1;
#pragma omp parallel private(x) num_threads(2)
{
   x++;
   printf("%d\n" , x);
}
printf("%d\n" , x);
```

x=1

Compartición de datos (*firstprivate*)

- Cuando una variable está definida como **firstprivate**, la variable se replica en la región paralela con una nueva variable del mismo tipo con un **valor inicializado al valor original**
  - Esto significa que todos los hilos tienen una variable diferente
  - Se puede acceder sin ningún tipo de sincronización

Compartición de datos

Imprime 2 (dos veces)

x=1

example_firstprivate.c

```
int x=1;
#pragma omp parallel firstprivate(x) num_threads(2)
{
   x++;
   printf("%d\n" , x);
}
printf("%d\n" , x);
```

Compartición de datos *por defecto*

- Variable estático/global es **shared**
- Variable en el *heap* es **shared**
- Variable asignado por pila dentro de la construcción es **private**
- Otros
  - Si existe la cláusula *default*, lo que dice la cláusula es:
    - **none** significa que el compilador emitirá un error si el atributo no es establecido explícitamente por el programador
    - De lo contrario, depende de la construcción
    - En la región paralela por defecto es **shared**

Compartición de datos *por defecto*

example_default.c

```
int x, y;
#pragma omp parallel private(y)
{
   x=
   y=
   #pragma omp parallel private(x)
   {
      x=
      y=
   }
}
```

x es *shared*

y es *private*

Compartición de datos *por defecto*

example_default.c

```
int x, y;
#pragma omp parallel private(y)
{
   x=
   y=
   #pragma omp parallel private(x)
   {
      x=
      y=
   }
}
```

x es *private*

y es *shared*

Compartición de datos *threadprivate*

```
#pragma omp threadprivate (var-list)
```

- Puede aplicar a
  - Variables Globales
  - Variables Estáticas
- Permite crear una copia de variables "globales" por hilo
- **treadprivate** persiste a los largo de las regiones **parallel** si el número de hilos es el mimso

Compartición de datos

example_threadprivate.c

```
char* foo( )
{
   static char buffer[ BUF_SIZE ];
   #pragm
   ...
   return
}
```

Crea una copia *estática* de **buffer** por hilo

Ahora **foo** puede ser invocada por multiples hilos al mismo tien

Construcción de trabajo compartido

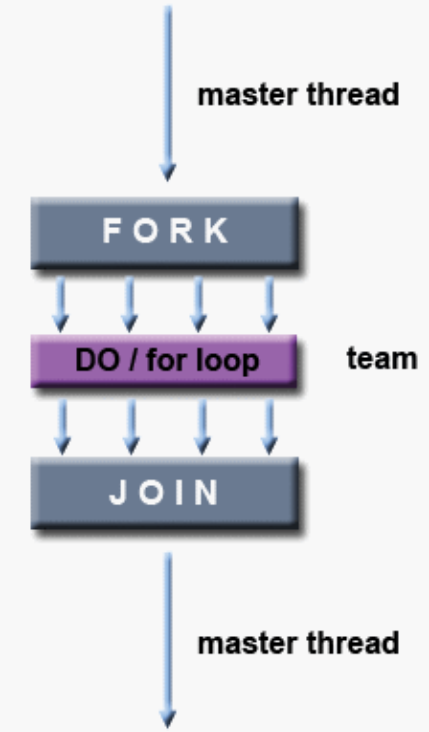
- Las **construcciones de trabajo compartido** dividen la ejecución de una región de código entre los hilos
  - Los hilos cooperan para hacer algún trabajo
  - Dividir el trabajo que usando ID de hilo
  - Menor sobrecarga que el uso de tareas
    - Pero, menos flexible

Tipos

- DO/FOR
- SECTIONS
- SINGLE

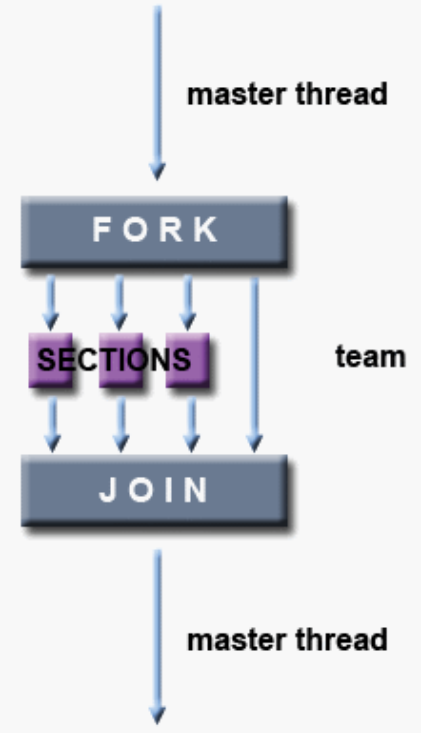
Construcción de trabajo compartido

- DO/FOR comparte las iteraciones de un bucle a lo largo del equipo de trabajo: **paralelismo de datos**



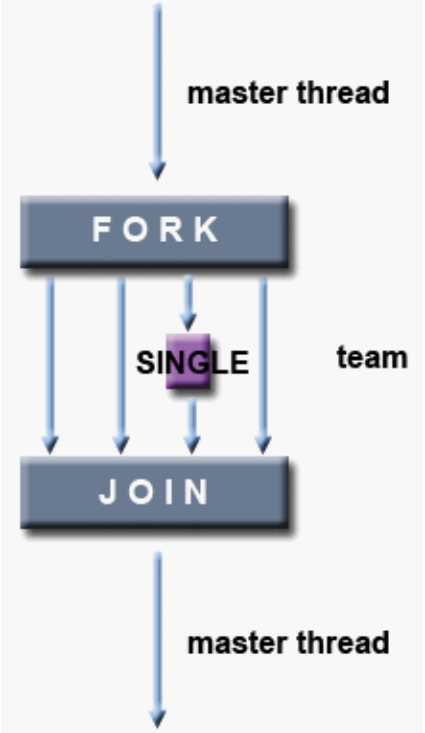
Construcción de trabajo compartido

- SECTIONS: rompe el trabajo en secciones separadas y discretas. Cada sección es ejecutada por un hilo: **paralelismo funcional**



Construcción de trabajo compartido

- SINGLE: serializa la sección del código



Paralelismo en bucles

- `#pragma omp for [clauses]`

Clausulas

- *private*
- *shared*
- *lastprivate*
- *firstprivate*
- *schedule*
- *nowait*
- *collapse*

Construcción `parallel for`

¿Cómo funciona?

- Las iteraciones del (los) bucle(s) se dividen
  - Entre los hilos del equipo.
  - **Las iteraciones de bucle deben ser independientes**
  - Los bucles deben construirse para conocer el número de iteraciones
    - Las variables de inducción son privatizadas automáticamente.
  - El atributo del tipo de datos por defecto es **shared**

Construcción parallel for

example_parallelfor.c

```
void foo(int *m, int N, int M)
{
   int i;

   #pragma omp parallel for private(j)
   for (i=0; i<N; i++)
      for(j=0; j<M; j++)
         m[i][j] = 0;
}
```

Implicitamente, la variable *i* es privada

Clausula `lastprivate`

- Cuando una variable es declarada **lastprivate**, la copia privada que permanece fuera de la región paralela es el valor correspondiente a la última iteración del bucle
  - Una variable puede ser al mismo timpo **firstprivate** y **lastprivate**

Clausula `reduction`

- Es bastante común que existan variables que acumulen valores
  - `red+=v[i]`, como suma de los valores de un vector
  - El uso de **critical** o **atomic** no es una buena solución desde el punto de vista del rendimiento
  - ... porque **serializa el cómputo**

`reduction`

- Operadores válidos: **+**, **-**, ***,|**, **||**, **&**, **&&**,**^**
- El compilador crea variables privadas (en cada hilo) y luego reduce las variables privadas
- También se permite la clausula *reduction* en la construcción **parallel**

Clausula reduction

example_reduction.c

```
int foo(int *vector, int N)
{
   int i;
   int sum=0;

   #pragma omp parallel for reduction(+:sum)
   for (i=0; i<N; i++)
      sum+=vector[i];

   return(sum);
}
```

Clausula `schedule`

- La cláusula de *schedule* permite determinar la distribución de ejecución de los hilos
  - Si la clausula no está presente, la implementación define cúal se escoje por defecto

Opciones

- *STATIC*
- *STATIC, chunk*
- *DYNAMIC[, chunk]*
- *GUIDED[, chunk]*
- *AUTO*
- *RUNTIME*

Clausula `schedule`

STATIC

- El total de iteraciones es distribuido de forma estática en *trozos* de igual tamaño
- Estos trozos o *chunk* se distribuyen según el algoritmo del Round-Robin
- Características
  - Bajo overhead en la planificiación
  - Buena localidad (normalmente)
  - Pueden existir desbalanceos de carga

Clausula `schedule`

STATIC N

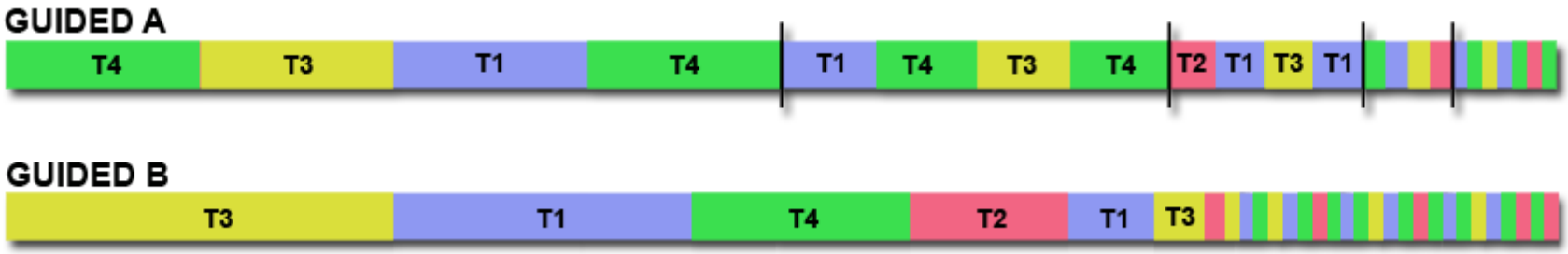
- Se dividen en *chunk* de tamaño N

Clausula `schedule`

DYNAMIC[, chunk]

- Se distribuyen de forma dinámica las iteraciones de *chunk* en *chunk* hasta acabar
  - Si *chunk* no especificado chunk=1
- Características
  - Alto sobrecarga en la planificación y no explota la localidad
  - ... **pero resuelve los problemas de deslabanceo**

Clausula `schedule`



GUIDED[, chunk]

- Variante de *dynamic*, el tamaño de los trozos disminuye con los hilos

Clausula `schedule`

AUTO

- La implementación de OpenMP permite fijar la planificación a su gusto

RUNTIME

- La decisión se retarda hasta el bucle de acuerdo a la variable **sched-nvar**
  - Mediante la variable de entorno `OMP_SCHEDULE`
  - O con la llamada a la API `omp_set_schedule()`

Clausula *nowait*

- En una construcción paralela implicitamente conlleva un **barrier** al final del bucle
- La clausula **nowait** la elimina
  - Permite la solapar ejecuciones **independientes** de bucles, tareas....

example_nowait.c

```
#pragma omp for nowait
for (i=0; i<N; i++)
   vector[i]=0;
#pragma omp for
for (i=0; i<N; i++)
   vector_out[i]=0;
```

Clausula *collapse*

- Permite distribuir el trabajo de bucles anidados
  - Evidentemente los bucles deben de ser independientes

example_collapse.c

```
#pragma omp for collapse(2)
for (i=0; i<N; i++)
   for (j=0; j<N; j++)
      vector_out[i][j]=0;
```

Construcción `sections`

- `#pragma omp sections`
  - Cada sección se distruye en los hilos disponibles
    - Conllevan barrera implícita

Clausulas

- *private*
- *lastprivate*
- *firstprivate*
- *reduction*
- *nowait*

Construcción sections

example_sections.c

```
#include<omp.h>
void foo()
{
   ...
   #pragma omp parallel sections num_threads(3)
   {
      #pragma omp section
      read(data);

      #pragma omp section
      #pragma omp parallel
      work(data); // Nested paralellism

      #pragma omp section
      write(data);
   }
}
```

Construcción `single`

- `#pragma omp single [clauses]`
  - Un hilo ejecuta una sección de código
  - Conlleva un **barrier** implícito

Clausulas

- *private*
- *firstprivate*
- *nowait*

Construcción single

example_single.c

```
int main()
{
   #pragma omp parallel
   {
      #pragma omp single
      {
         printf("Hello, I am %d!!!\n", get_thread_num());
      }
   }
}
```

Solo un hilo imprime