OpenMP

Grado en Ingeniería Informática

Programación Paralela con Memoria Compartida: OpenMP

Departamento de Ingeniería Informática
Universidad de Cádiz
Pablo García Sánchez

Apuntes basados en el trabajo de Guadalupe Ortiz y en el de Mattson y Meadows

Curso 2016 – 2017

Indice

1. Introducción
2. Componentes de OpenMP: Directivas, funciones y variable de entorno
3. Regiones y secciones paralelas
4. Cláusulas
5. Bucles
6. Sincronización
7. Esquemas Algorítmicos Paralelos

Sección 1 | Introducción

¿Qué es OpenMP? (I)

- Open Multi-Processing
- Es un modelo multi-hebra de variables compartidas: Las hebras se comunican compartiendo variables.
- Compartición no intencionada de datos puede producir condiciones de carrera: cuando la salida del programa cambia si las hebras son distribuidas de manera distinta.
- Para controlar condiciones de carreara: usar sincronización para protejer conflictos de datos.

¿Qué es OpenMP? (II)

API para programas paralelos multithread en memoria compartida

- Directivas de compilación
- Funciones
- Variables de entorno

Conceptos básicos

- Utiliza el modelo fork-join
- El programador tiene control explícito sobre la paralelización
- Hay secciones secuenciales y secciones paralelizables

Paralelismo fork-join

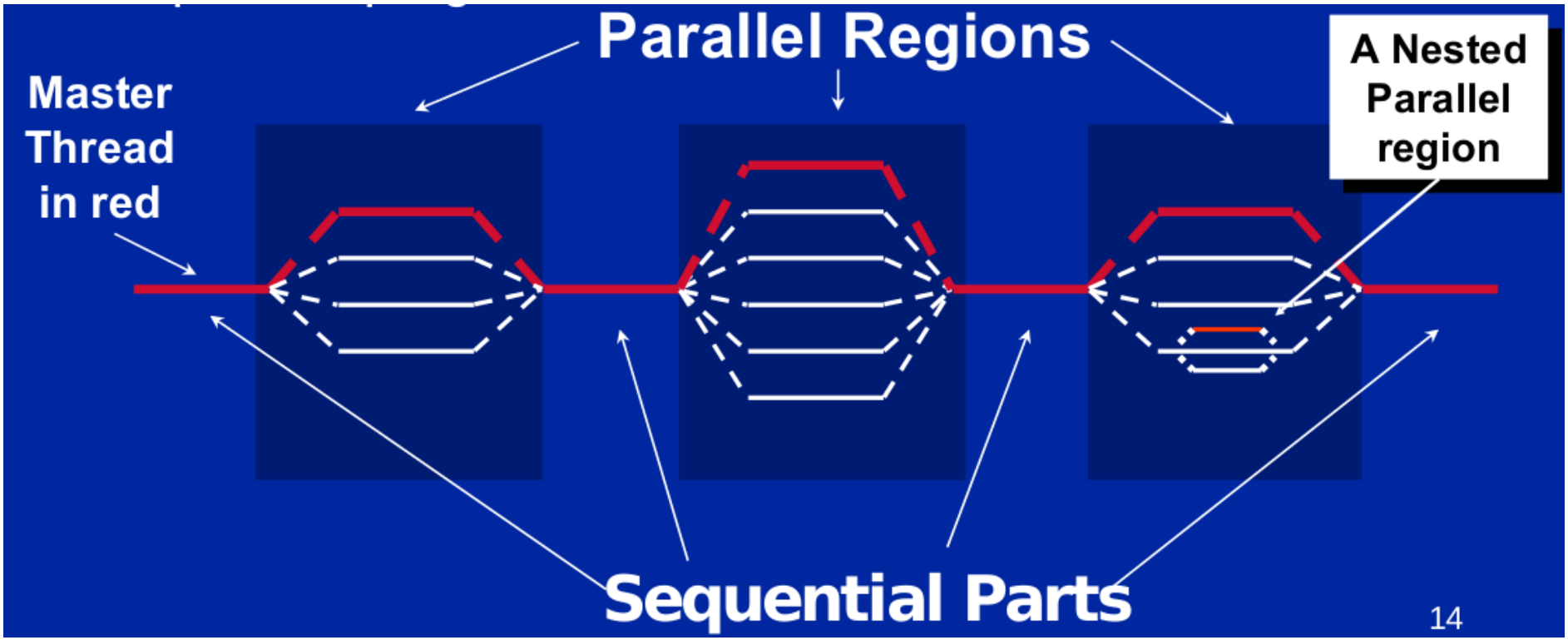

Figura: Extraída de la presentación de OpenMP de Mattson y Meadows.

Introducción (II)

Modelo de programación

- Directivas: `#pragma omp <directiva OpenMP>`
- Un compilador secuencial ignora estas directivas
- Cada thread paralelo puede ejecutarse en un procesador, aunque no se garantiza

Sección 2 | Componentes de OpenMP: Directivas, funciones y variable de entorno

Directivas

El preprocesador del compilador las sustituye por código.

Convenciones sobre las directivas

- Son sensibles a las mayúsculas/minúsculas
- Sólo puede especificarse un nombre de directiva por directiva
- Cada directiva se aplica al menos a la sentencia que le sigue
- Las directivas largas pueden continuarse usando \al final de línea
- Formato general:
  `#pragma omp nombre-de-directiva [cláusulas, ...]`
  `nueva línea`
- Ejemplo: `#pragma omp parallel num_threads(8) if(N>20)`

Funciones de la librería

- `void omp_set_num_threads (int num_threads)` Tiene prioridad sobre la variable de entorno OMP_NUM_THREADS
- `int omp_get_num_threads (void)` número de threads actualmente en ejecución
- `int omp_get_max_threads (void)` máximo valor que puede devolver get_num_threads
- `int omp_get_thread_num (void)` el maestro es el cero
- `int omp_get_num_procs (void)` procesadores físicos disponibles

Variables de entorno

Variables de entorno

- OMP_NUM_THREADS
  `setenv OMP_NUM_THREADS 4`
- OMP_SCHEDULE sólo se aplica en los bloques con cláusula schedule a runtime
  `setenv OMP_SCHEDULE “guided, 4”`
- OMP_DYNAMIC activa/desactiva el ajuste dinámico de número de threads
  `setenv OMP_DYNAMIC TRUE`
- OMP_NESTED: activa/desactiva el paralelismo anidado
  `setenv OMP_NESTED TRUE`

Ejemplo: g++ -fopenmp ejemplo.cpp -o ejemplo

```
#include "omp.h"
#include "stdio.h"

int main(){
#pragma omp parallel
  {
    int ID = omp_get_thread_num();
    printf("Hola, soy la hebra %d\n",ID);
    printf("Adios, soy la hebra %d\n",ID);
  }
 return 0;
}
```

Sección 3 | Regiones y secciones paralelas

Regiones paralelas

```
#pragma omp parallel [clausulas..]
bloque estructurado
```

- Claúsulas: if, private, shared , default, firstprivate, reduction, copyin
- Cuando un thread llega a una región paralela expande un grupo de threads y se convierte en el maestro (número 0). El número de threads se determina según la siguiente prioridad:
  - omp_set_num_threads
  - OMP_NUM_THREADS
- Sólo el thread maestro continúa la ejecución al final de la región paralela

Secciones

- La directiva sections especifica qué código de los bloques agrupados por la directiva se dividirá entre los threads del equipo.
- Tiene que estar en una región paralela
- Barrera implícita al final
- Puede haber threads que ejecuten más de una sección

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

- Cláusulas: private, firsprivate, lastprivate, reduction, nowait

Ejemplo de secciones

```
#include <stdio.h>
#include <omp.h>
void funcA() {
  printf("En␣funcA:␣esta␣seccia␣ejecuta␣el␣thread␣%d\n",
  omp_get_thread_num());
}
void funcB() {
  printf("En␣funcB:␣esta␣seccin␣la␣ejecuta␣el␣thread␣%d\n",
  omp_get_thread_num());
}
main() {
  #pragma omp parallel
  {
    #pragma omp sections
    {
      #pragma omp section
      (void) funcA();
      #pragma omp section
      (void) funcB();
    }
  }
}
```

Combinación de directivas

- Las cláusulas pueden combinarse en una
- `#pragma omp parallel for [cláusulas...]`
  `bucle for` Cláusulas: las de parallel y las de for
- `#pragma omp parallel sections[cláusulas...]`
  `secciones` Cláusulas: las de parallel y las de sections

Sección 4 | Cláusulas

Paralelización condicionada

if (expresión-escalar). Sólo puede usarse un if por directiva paralela.

```
#pragma omp parallel for [...] if (condición)
//región paralela
```

Grado de concurrencia

- `num_threads` (expresión-entera)
- `nowait` (los threads del bucle paralelo no se sincronizan al final de este)
- `ordered` (especifica secciones que deben ejecutarse en cierto orden secuencial)
- `single, master...`

Gestión de datos

- `private` (lista de variables) local a cada thread (no se inicializan)
- `firstprivate` (lista de variables) se inicializan al entrar en los threads con los valores anteriores a la directiva paralela
- `lastprivate` (lista de variables) private con una copia del valor en la última iteración paralela
- `copyin` (lista de variables) mecanismo para asignar el mismo valor a variables threadprivate al comienzo de la ejecución
- `shared` (lista de variables) todos los threads acceden a la misma posición de memoria
- `default` (shared | private | none) ámbito por defecto para todas las variables
- `reduction` (operator: lista de variables) cómo se combinan todas las variables locales de cada thread en el thread maestro

Gestión de datos y recursos (II)

`schedule (tipo [,tamaño]`: asignación de iteraciones a threads:

- `static`: se divide en bloques de tamaño y se asignan con round- robin. Si no se especifica tamaño se divide en tantos bloques como threads.
- `dynamic`: se divide en bloques de tamaño y se asignan tan pronto como un thread quede ocioso. Si no se especifica tamaño los bloques contienen una iteración.
- `guided`: cuando un thread computa un bloque se le asigna dinámicamente el siguiente bloque. Va reduciendo el tamaño del bloque (primer valor dependiente de la implementación). El número de iteraciones en el bloque se reduce hasta el valor tamaño.
- `runtime`: La variable de entorno OMP_SCHEDULE debe contener el tipo de asignación. No se especifica tamaño.

threadprivate

threadprivate

- Directiva para tipo de datos threadprivate hace que las variables globales sean locales y persistentes a los threads a lo largo de varias regiones paralelas

Sección 5 | Bucles

Ejemplo: Sumar dos vectores

Código secuencial

```
for(i=0;i<N;i++){a[i] = a[i]+b[i]}
```

OpenMP Parallel

```
#pragma omp parallel
{
 int id, i, Nthrds, istart, iend;
 id = omp_get_thread_num();
 Nthrds = omp_get_num_threads();
 istart=id*N/Nthrds;
 iend = (id+1)*N/Nthrds;
 if(id == Nthrds-1) iend = N
 for(i=istart;i<iend;i++){a[i]=a[i]+b[i];}
}
```

Más fácil

Constructo para reparto de trabajo

```
#pragma omp parallel
#pragma omp for
for(i=0;i<N;i++){a[i] = a[i]+b[i]}
```

- La variable i se hace "private"para cada hebra por defecto
- Se podría hacer explícitamente con la cláusula `private(i)`

Bucles en paralelo: restricciones

- Debe estar dentro de una región paralela
- Las iteraciones deben ser ejecutadas en paralelo por todos los threads
- Se supone que hay independencia de datos entre las interacciones
- Debe fijarse `schedule` a un entero invariante en el bucle
- Cláusulas: `schedule`, `ordered`, `private`, `shared`, `firsprivate`, `lastprivate`, `shared`, `reduction`, `nowait`

Ejemplo de planificación

```
#pragma omp parallel for private(j) schedule (dynamic,5)
for( i= 0 ; i< 5000 ; i++ )
  for( j= 0 ; j< f(i); j++ )
    tarea_costo_variable(i,j);
```

Trabajando con bucles

Secuencial

```
int i,j, A[MAX];
j = 5;
for(i=0;i<MAX;i++){
 j += 2;
 A[i] = func(j);
}
```

Paralelo

```
int i, A[MAX];
#pragma omp parallel for
for(i=0;i<MAX;i++){
 int j = 5+2*i;
 A[i] = func(j);
}
```

- El índice i es privado por defecto
- Las iteraciones deben hacerse independiente, para que puedan ejecutarse en cualquier orden sin inter dependencias al bucle.

Reducción

¿Cómo manejamos este caso?

```
double media = 0.0, A[MAX]; int i;
for(i = 0; i<MAX;i++){
  media += A[i];
}
media = media/MAX;
```

- Estamos combinando valores en una única variable de acumulación (media), pero hay dependencia entre iteraciones que no es trivial eliminar.
- Situación común: reducción

Reducción (II)

- `reduction (op:lista)`
- Se hace una copia local de cada variable de lista y se inicia dependiendo de "op" (ejemplo: 0 para +).
- El compilador encuentra las expresiones que contengan "op" y las usa para actualizar la copia local.
- Las copias locales se reducen a un valor único y se combinan al valor global original.

```
double media = 0.0, A[MAX]; int i;
#pragma omp parallel for reduction (+:media)
for(i = 0; i<MAX;i++){
  media += A[i];
}
media = media/MAX;
```

Sección 6 | Sincronización

Sincronización (I)

- Directiva `single`: especifica que el código será ejecutado por un único thread

  ```
  #pragma omp single [cláusulas...]
  bloque estructurado
  ```

  - Cláusulas: private, firsprivate, nowait
- Directiva `master`: especifica que el código será ejecutado por el thread maestro. Es recomendable poner barreras al terminar.

  ```
  #pragma omp master
  bloque estructurado
  ```

- Directiva `critical`: especifica que el código será ejecutado sólo por un thread a la vez

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

- Las regiones con el mismo nombre son tratadas como la misma

Sincronización (II)

- La directiva `atomic`: especifica que una posición específica de memoria debe ser modificada de forma atómica. Se usa con escalares, expresiones escalares y operaciones binarias

```
#pragma omp atomic
sentencia
```

- Directiva `barrier`: sincroniza todos los threads

```
#pragma omp barrier
```

- Directiva `ordered`: el código se ejecuta en el orden en el que las iteraciones hubieran sido ejecutadas en una ejecución secuencial
- Directiva `flush`: identifica un punto de sincronización con una visión consistente de la memoria

```
#pragma omp flush [lista de variables]
bloque estructurado
```

- En muchas directivas hay un flush implícito

Ejemplo barrier

```
#include <omp.h>
#include <stdio.h>
int main( ){
  int a[5], i;
  #pragma omp parallel
  {
  #pragma omp for
  for (i = 0; i < 5; i++)
    a[i] = i * i;
  // Maestro imprime resultados intermedios
  #pragma omp master
  for (i = 0; i < 5; i++)
    printf("a[%d] = %d\n", i, a[i]);
  #pragma omp barrier
  #pragma omp for
  for (i = 0; i < 5; i++)
    a[i] += i;
  }
}
```

Sección 7 | Esquemas Algorítmicos Paralelos

Maestro/esclavo

- Proceso llamado maestro, pone en marcha a otros procesos llamados esclavos.
- Los maestros asignan trabajos y recaban soluciones parciales de los esclavos.
- OpenMP trabaja inicialmente con un único thread (Maestro) y al llegar a un constructor parallel pone en marcha los thread esclavos.
- Al acabar la ejecución hay una sincronización implícita.

M/E con Número fijo de trabajos y asignación estática

Trabajos de mismo coste

- Si se dispone de suficientes procesadores se utiliza un proceso (thread) por cada trabajo.
- Si no se dispone de suficientes procesadores se agrupan en bloques.

Trabajos de diferente coste

- Se agrupan en bloques con coste similar (en caso de disponer de sistemas homogéneos.)

M/E con Número fijo de trabajos y asignación dinámica

- Se asigna un trabajo a cada procesador hasta que finaliza y se le asigna un nuevo trabajo.
- Cuando un esclavo finaliza un trabajo informa al maestro para que le asigne otro.
- No es necesario obtener una distribución equilibrada antes de comenzar.

Maestro esclavo

t trabajos y p procesos

Maestro

```
asignar un trabajo a cada esclavo;
t=t-p;
while(t!=0){
  Recibe(resultado,esclavo);
  esclavo.Envia(trabajo)
  t=t-1;
}

while(p!=0){
 Recibe(resultado,esclavo);
 esclavo.Envia(TERMINAR);
 p=p-1;
}
```

Esclavo

```
Maestro m;
m.Recibe(trabajo);
resolver trabajo;
m.Envia(resultado);
m.Recibe(trabajo o TERMINAR);


while(no TERMINAR){
 resolver trabajo;
 m.Envia(resultado);
 m.Recibe(trabajo o TERMINAR);
}
```

M/E con Número de trabajos desconocido

- Se generan nuevos trabajos durante la computación.
- Los esclavos realizan las tareas e indican al maestro los resultados y las nuevas tareas.
  - Cuando no puede atender las peticiones el esclavo espera.
  - Cuando al maestro le llegan nuevas tareas se las manda a los esclavos en espera.
- El maestro puede gestionar una bolsa de tareas.

M/E con tamaño de trabajos desconocido

t trabajos y p procesos

Maestro

```
generar tareas iniciales;
asignar un trabajo a cada esclavo;
solicitudes = 0;
while(solicitudes!=p){
  Recibe(resultado trabajos o solicitud);

  if(recibidos trabajos){
    for(cada esclavo esperando){
      if(trabajo disponible){
        Envia(trabajo, esclavo);
        solicitudes = solicitudes - 1;
      }
    }
  }

  if(recibida solicitud){
    solicitudes = solicitudes +1;
    if(trabajo disponible)
      Envia(trabajo, esclavo recibido);
  }
}
enviar TERMINAR a todos los esclavos
```

Esclavo

```
Maestro m;
m.Recibe(trabajo);
resolver trabajo;
m.Envia(resultado);
m.Recibe(trabajo o TERMINAR);


while(no TERMINAR){
 resolver trabajo;
 m.Envia(resultado);
 m.Recibe(trabajo o TERMINAR);
}
```

Trabajadore replicados

- Conjunto de procesadores (trabajadores) colaborando en la resolución de un problema
- Parecida al maestro-esclavo.
- Los trabajadores replicados trabajan de manera autónoma, resolviendo tareas que dan lugar a otras tareas.
- La condición de fin es que no queden tareas en la bolsa y no haya trabajadores trabajando.
- Se puede asignar un proceso para la gestión de la bolsa de tareas (maestro/esclavo).
- Es un buen método cuando no se conoce el número de tareas o su coste o con procesadores heterogéneos

Trabajadores replicados

```
Inicializar bolsa de tareas;
Inicializar tareas;
omp_set_num_threads(trabajadores);
acabados = trabajadores;
#pragma omp parallel private(fin,n)
fin=false;
while(!fin){
  #pragma omp critical{
    if(tareas = 0 &&
      acabados = trabajadores){
       fin=true;
    else
     if(tareas != 0 ){
       acabados = acabados - 1;
       tareas = tareas -1;
       tomar tarea de la bolsa;
     }
  } //fin critical
```

```
  if(tomada tarea){
    (resolver tarea y
    generar n nuevas tareas;)
    #pragma omp critical{
     tareas = tareas + n;
     acabados = acabados + 1;
     (insertar las n tareas
      en la bolsa;)
    }
  }
} //end while
} //end parallel
```

- Se accede en exclusión mutua a la bolsa de tareas y a las variables de control.
- Tareas: número de tareas en la bolsa.
- Trabajadores: número de trabajadores replicados.
- Acabados: número de trabajadores que han acabado.

Referencias

- Almeida, F., Giménez, D., Mantas, J.M., Vidal, A.M. Introducción a la Programación Paralela. Editorial Paraninfo, 2008.
- Apuntes de Guadalupe Ortiz
- Apuntes de Julio Ortega y Mancia Anguita
- Presentación de Mattson y Meadows