Tema 4.5 Programación mediante directivas OpenMP: Tareas

Computación de Altas Prestaciones

Carlos García Sánchez

24 de octubre de 2022

- "Using OpenMP : portable shared memory parallel programming", Barbara Chapman, et all. 2008
- "OpenMP 5.2", https://www.openmp.org/wp-content/uploads/OpenMP-API-Specification-5-2.pdf

Outline

¿Qué es una tarea?

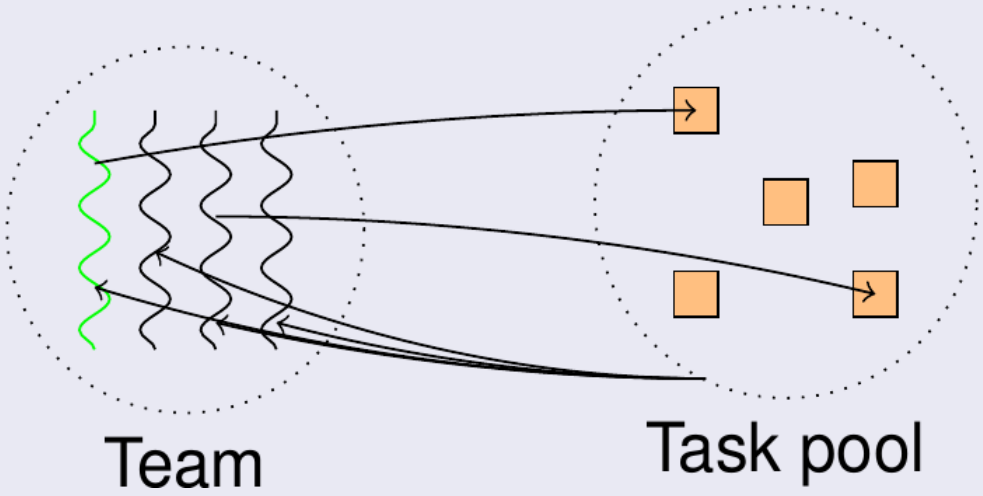
- Tareas son unidades de trabajo cuya ejecución
  - puede ser diferidas o...
  - ... se puede ejecutar inmediatamente
- Tareas se componen de
  - **código** a ejecutar, los **datos**, variables de control **interno** (ICVs)
- Las tareas son creadas...
  - ... al llegar a una región paralela: tareas implícitas (por hilo)
  - ... cuando aparece una construcción *task*: tareas explícitas
  - ... cuando aparece una construcción *taskloop*: tareas explícitas por *chunk*
  - ... cuando aparece una construcción *target*: se crea la tarea de destino

Modelo de ejecución

- Soporta paralelismo no-estructurado
  - Bucles sin límites definidos: `while ( <expr> ) {...}`
  - Funciones recursivas: `void myfunc( <args> ){...; myfunc( <newargs> ); ...;}`
- Varios escenarios posibles
  - Creador único, multiples creadores, tareas anidadas...
- Todos los hilos de un "equipo" son candidatos a ejecutar una **tarea**

Modelo de paralelismo de tareas con OpenMP

Task parallelism



- Paralelismo extraido en secciones de código independiente
- Permite para explotar paralelismo no-estructurado
  - Bucles sin límites, funciones recursivas

Modelo de paralelismo de tareas con OpenMP

¿Que es un tarea en OpenMP?

- Tareas = unidades de trabajo (ejecución puede diferirse)
  - También pueden ser ejecutados inmediatamente
- Las tareas se componen de:
  - código para ejecutar
  - un entorno de datos
    - Inicializado en el momento de la creación
  - variables de control interno (ICVs)
- Hilos pueden **cooperar** para ejecutarlas

Construcción `task`

- `#pragma omp task [clauses]`

Clausulas Datos

- *shared*
- *private*
- *firstprivate*
- *default(shared|none)*
- *in_reduction*

Sincronización

- *depend(dep-type:list)*

¿Cuando se crean las tareas?

- En una región **parallel**
  - Una tarea implicita se crea para ser asignada al thread
- Cada hilo que entra en la construcción **task**
  - Necesita el código y datos para esa tarea
  - Se crea una nueva tarea expecíficamente

Compartición de datos entre tareas

Si no hay clausula expecífica

- Se usan las reglas por defecto
  - Ej: variables globales como *shared*, inicializadas antes de la contrucción como *firstprivate*

example_task.c

```
int a;
void foo(){
   int b, c;
   #pragma omp parallel shared(c)
   #pragma omp parallel firstprivate(b)
   {
      int d;
      #pragma omp task
      {
         int e;
         a = //shared
         b = //firstprivate
         c = //shared
         d = //firstprivate
         e = //private
      }
}}
```

Planificación de tareas: tied vs untied tasks

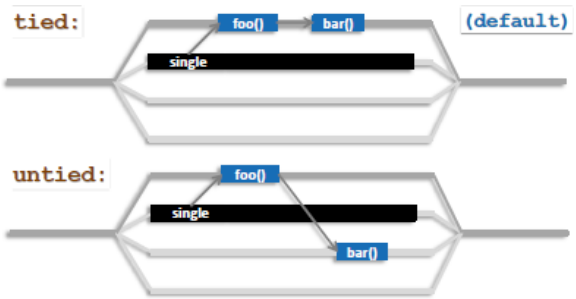
- Las tareas son del tipo *tied* por defecto
  - Estas tareas siempre se ejecutan por el mismo hilo
  - Pero puede haber problemas en tareas con características basadas en hilos thread-id, regiones críticas...
- El programador puede especificar tareas como *untied* (planificación relajada)
  - `#pragma omp task untied`
    - Pueden migrar entre hilos

Tied vs untied tasks

- Para indicar puntos de planificación se usa directiva *taskyield*
  - Tareas se pueden suspender (algunos requisitos extra evitan deadlocks)
  - Se puede indicar esplícitamente con directiva `#pragma taskyield`

taskyield.c

```
#pragma omp parallel
#pragma omp single
{
   #pragma omp task untied
   {
      foo();
      #pragma omp taskyield
      bar()
   }
}
```



Sincronización de tareas

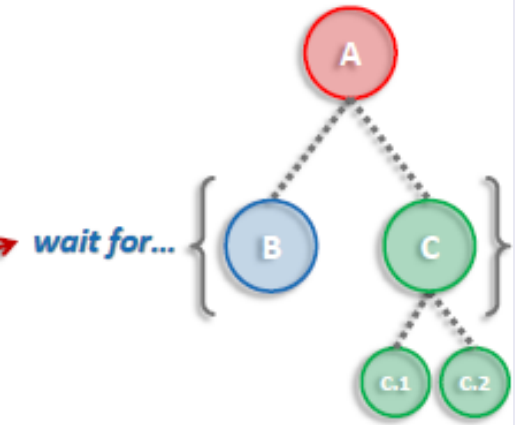
- Existen dos contrucciones básicas:
  - **barrier**: esperar hasta que lo anterior se complete
  - **taskwait**

Construcción `taskwait`

`#pragma omp taskwait`

- Suspende la tareas hasta que todos los **hijos** completen la tarea

```
#pragma omp parallel
#pragma omp single
{
  #pragma omp task                    :A
  {
    #pragma omp task                  :B
    { … }
    #pragma omp task                  :C
    { … #C.1; #C.2; …}
    #pragma omp taskwait
  }
} // implicit barrier will wait for C.x
```



Reducciones

- Operación de reducción
- Dos directivas
  - `#pragma omp taskgroup task_reduction(op: list)` registra una reducción en **[1]**, y computa el resultado final después de **[3]**
  - `#pragma omp task in_reduction(op: list)` para participar en la operación de reducción en **[2]**

reduction.c

```
int res = 0;
node_t* node = NULL;
...
#pragma omp parallel
{
  #pragma omp single
  {
    #pragma omp taskgroup task_reduction(+: res)
    { // [1]
      while (node) {
        #pragma omp task in_reduction(+: res) \
          firstprivate(node)
        { // [2]
          res += node->value;
        }
        node = node->next;
      }
    } // [3]
  }
}
```

Planificación tareas *if*

- Si la cláusula *if* se evalúa como falsa
  - Se suspende la tarea
  - La nueva tarea **se ejecuta inmediatamente**
    - con su propio entorno de datos
    - diferente tarea con respecto a la sincronización.

API para Tareas asíncronas

- Existen API para explicitar tareas asíncronas en otros paradigmas de programación como MPI, HIP, CUDA....
- Los programadores de OpenMP necesitan algún mecanismo para desarrollar un programación parecida

async.c

```
do_something();
hipMemcpyAsync(dst, src, nbytes, hipMemcpyDeviceToHost, stream);
do_something_else();
hipStreamSynchronize(stream);
do_other_important_stuff(dst);
```

API para OpenMP

- **Problemática**: condición de carrera entre tarea A y C
  - Tarea C puede comenzar antes que concluya la tarea A

async_openmp.c

```
void hip_example() {
    #pragma omp task
    { // task A
        do_something();
        hipMemcpyAsync(dst, src, nbytes, hipMemcpyDeviceToHost, stream);
    }
    #pragma omp task
    { // task B
        do_something_else();
    }
    #pragma omp task // task C
    {
        hipStreamSynchronize(stream);
        do_other_important_stuff(dst);
    }
}
```

API para OpenMP

- **Problemática**: dependencias entre operaciones
  - La tarea C estará bloqueada hasta que la A se complete

async_openmp.c

```
void hip_example() {
    #pragma omp task depend(out:stream)
    { // task A
        do_something();
        hipMemcpyAsync(dst, src, nbytes, hipMemcpyDeviceToHost, stream);
    }
    #pragma omp task
    { // task B
        do_something_else();
    }
    #pragma omp task depend(in:stream) // task C
    {
        hipStreamSynchronize(stream);
        do_other_important_stuff(dst);
    }
}
```

Tareas "detachable"

- Desde OpenMP 5.0 se introduce el concepto de tareas "detachable"
  - La tarea puede separarse del subproceso en ejecución sin "completarse"
  - Se pueden aplicar mecanismos regulares de sincronización de tareas para esperar la finalización de un tarea separada

API de tiempo de ejecución para completar una tarea:

- Eventos para tareas "detachable": `omp_event_t datatype`
- Clausula para tareas "detachable": `detach(event)`
- API del runtime: `void omp_fulfill_event(omp_event_t *event)`

Tareas "detachable"

1. Tarea "detachable" que se queda a la espera
2. La construcción *taskwait* no puede ejecutarse hasta que no complete la tarea
3. Envío del "evento" para proseguir con la tarea
4. Tarea completa, hace el *taskwait* y puede continuar

```
omp_event_t *event;
void detach_example() {
#pragma omp task detach(event)
    {
        important_code();
    }①

    #pragma omp taskwait ② ④
}
```

Some other thread/task:

```
omp_fulfill_event(event); ③
```

Tareas "detachable"

- Uniendo todo

async_openmp.c

```
void callback(hipStream_t stream, hipError_t status, void *cb_dat) {
   omp_fulfill_event((omp_event_t *) cb_data);
}

void hip_example() {
   omp_event_t *hip_event;
   #pragma omp task detach(hip_event)
   { // task A
      do_something();
      hipMemcpyAsync(dst, src, nbytes, hipMemcpyDeviceToHost, stream);
      hipStreamAddCallback(stream, callback, hip_event, 0);
   }
   #pragma omp task
   { // task B
      do_something_else();
   }

   #pragma omp taskwait
   #pragma omp task
   {  // task C

      do_other_important_stuff(dst);
   }
}
```