Constructos de Sincronización y Paralelismo de Tareas

CC3069 – Computación Paralela y Distribuida

Agenda

- Constructos de Sincronización
  - Critical, Atomic y Barrier
  - Master y Single
  - Otros Constructos
- Paralelismo de tareas
  - Directivas sections / section
  - Directiva task

Constructos de Sincronización en OpenMP

Directivas MutEx

- Las siguientes directivas, relacionadas con el propósito de exclusión mutua, aseguran que ciertos bloques de código se ejecuten como secciones críticas:
  - #pragma omp critical [(identifier)]{/*struct*/}
  - #pragma omp atomic
  - #pragma omp master
  - #pragma omp single
  - Lock Explicitos

#pragma omp critical [(ident)]

- Indica que el siguiente bloque de código se accede por un thread a la vez. El identificador es opcional.
- Si no tiene identificador, OpenMP considera todos esos bloques como el mismo bloque crítico.
- Usar identificadores y nombrar distintas secciones críticas puede ayudar a obtener mejor performance, al permitir ejecución concurrente de secciones críticas disjuntas.

#pragma omp atomic

- Versión más “ligera” que la directiva *critical* que usa instrucciones atómicas de máquina (específicas a la plataforma, Intel, AMD etc).
- Utilizado para instrucciones simples que son críticas. Solamente acepta una instrucción la cual además debe cumplir ciertos requisitos.

#pragma omp atomic

- Operaciones posibles que pueden ser atomic:
  - x++, x- -, ++x, - -x
  - x **binop**= expr;
  - x = x **binop** expr;
  - x = expr **binop** x;
- **Binop** := {+, *, -, /, &, |, <<, >>}
- expr := scalar expression
- Consideraciones:
  - Aunque update a x es atomico...
  - Y++ no lo es y puede causar race condition
  - En ese caso mejor usar critical

```
#pragma omp atomic
    x += y++;
```

#pragma omp master

- Obliga a que solamente el thread maestro ejecute el bloque de instrucciones siguiente.
- Otros threads se saltan ese bloque y proceden con el flujo del programa.
- No tiene barrera implícita en entrada ni en salida.
- Usado para cosas como:
  - Operaciones I/O
  - Coordinación y “message-passing”

#pragma omp single

- Obliga la ejecución del bloque a que sea ejecutada por un thread solamente (cualquiera).
- Si tiene barrera implícita al final
- Salvo se empareje con un 'nowait' (#pragma omp single nowait)
- Usualmente se emplea junto/dentro de un constructo *parallel*, para limitar ciertas partes a ejecutarse secuencialmente.

#pragma omp single

Por ende los siguientes son equivalentes:

- #pragma omp single {}
- #pragma omp master{}
  #pragma omp barrier

```
double data[ N ];
#pragma omp parallel shared( data, N )
{

#pragma omp single
  {
    // read data from a file
  }

#pragma omp for
  for(int i = 0; i < N; i++)
    {
       // process the data
    }
}
```

Locks Explícitos

- OpenMP provee mecanismos de lock para sincronización de threads.
- Pueden ser 'simple' o 'nestable' (anidados)
- Los métodos para anidados se llaman igual que los simples, excepto porque llevan 'nest_lock()'

```
void omp_init_lock(omp_lock_t*    lock_p   /* out */);
void omp_set_lock(omp_lock_t*     lock_p   /* in/out */);
void omp_unset_lock(omp_lock_t*   lock_p   /* in/out */);
void omp_destroy_lock(omp_lock_t* lock_p   /* in/out */);
```

Locks Explícitos

- Los locks de OpenMP solo deben ser accesados por las rutinas propias de OpenMP.
- El ciclo de vida de los locks es:
  - Inicializar (init). Inicia desbloqueado por defecto.
  - Utilizar (set, unset). Bloquea y Desbloquea el lock.
  - Remover (destroy)

```
void omp_init_lock(omp_lock_t*    lock_p   /* out */);
void omp_set_lock(omp_lock_t*     lock_p   /* in/out */);
void omp_unset_lock(omp_lock_t*   lock_p   /* in/out */);
void omp_destroy_lock(omp_lock_t* lock_p   /* in/out */);
```

Critical? Atomic? Locks?

- Resulta natural pensar en qué casos conviene o es lo correcto utilizar cada uno de los constructos mencionados:
  - Atomic tiene potencial de ser el método mutex más rápido. Si solo se tiene una sección crítica, y cumple con los requisitos de atomic es mejor que usar critical.
  - Si la parte crítica es un bloque, o bien si se tienen varias partes críticas, conviene utilizar critical (o locks) en vez.
  - En general, se recomienda utilizar locks para casos donde la exclusión mutua se requiere en una estructura de datos, y no en un bloque de código.

Algunas Consideraciones

- Se recomienda no mezclar criticals con atomics protegiendo las mismas secciones o variables. Si no se puede reescribir la parte #critical de forma que cumpla lo requerido por #atomic entonces se debe pasar todo a #critical.
- Anidar constructos mutex puede ser problemático. En algunos casos puede resultar en un deadlock.

```
#  pragma omp critical
   y = f(x);
   . . .
   double f(double x) {
#     pragma omp critical
      z = g(x);  /* z is shared */
      . . .
   }
```

```
#  pragma omp critical(one)
   y = f(x);
   . . .
   double f(double x) {
#     pragma omp critical(two)
      z = g(x);  /* z is global */
      . . .
   }
```

Directivas de Sincronización

- Las siguientes directivas, relacionadas con el propósito de sincronización de eventos, se encargan de temas como la coordinación, consistencia de datos y ordenamiento de operaciones:
  - #pragma omp barrier
  - #pragma omp taskwait
  - #pragma omp ordered
  - #pragma omp flush [(list_of_variables)]

#pragma omp barrier

- Barrera explicita que asegura que todos los threads del equipo lleguen a la barrera antes de continuar.

```
double data[ N ];
#pragma omp parallel shared( data , N )
{

#pragma omp master
  {
    // read data from a file
  }

#pragma omp barrier

#pragma omp for
  for(int i = 0; i < N; i++)
    {
       // process the data
    }
}
```

#pragma omp ordered

- Utilizado en conjunto y dentro de parallel for.
- Obliga a ejecutar esa parte de forma secuencial, incluso si usamos constructos de scheduling.
- 'ordered' va en el for pragma. Y dentro del for se especifica el bloque *ordered*.

```
double data[ N ];
#pragma omp parallel shared( data , N )
{

#pragma omp for ordered schedule( static , 1 )
  for(int i = 0; i < N; i++)
    {
        // process the data

        // print the results in order
#pragma omp ordered
        cout << data[i];
    }
}
```

#pragma omp flush [(listOfVar)]

- Un constructo bastante interesante, el cual nos asegura una visión consistente de la memoria.
- Se le suele describir como una “barrera de memoria”.
- Garantiza que las instrucciones siguientes al *flush* sucedan hasta que las modificaciones de los registros/caché se propaga a memoria principal.
- Si no se incluye lista de variables, flush aplica a todas.

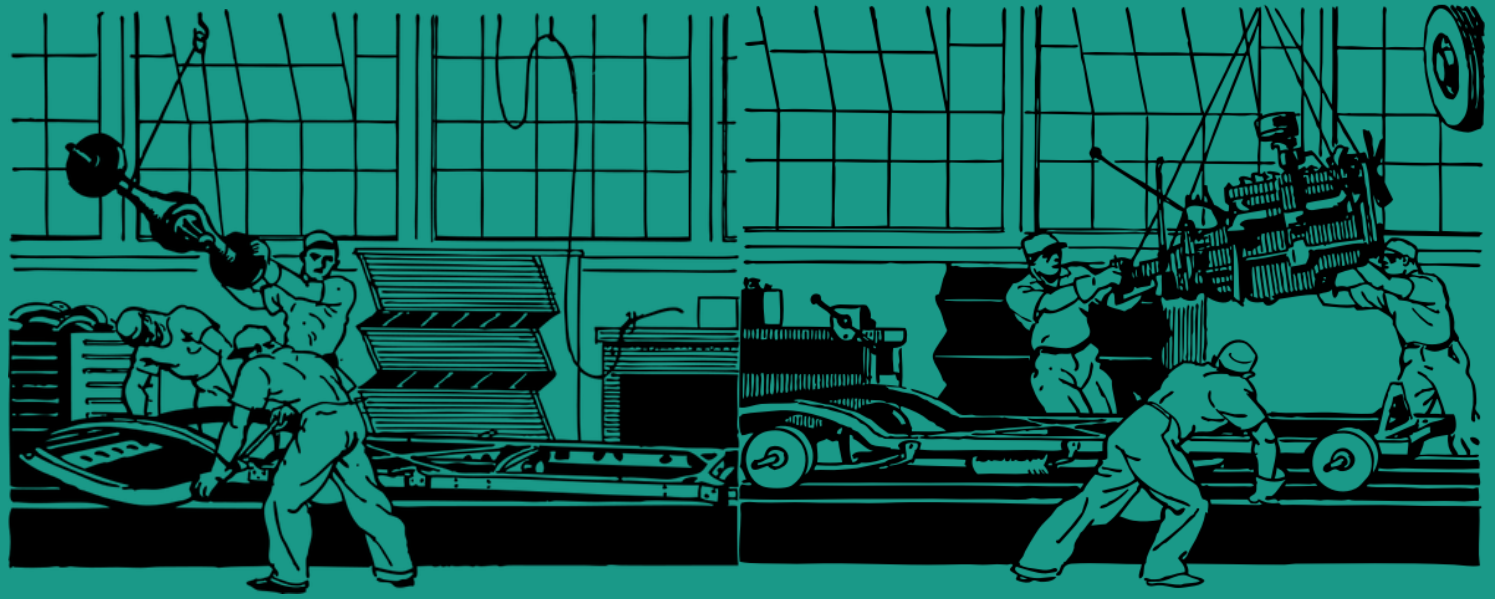
#pragma omp flush [(listOfVar)]

- Así como a veces suceden barreras implícitas, hay casos donde suceden flush implícitos:
  - al utilizar una directiva #barrier
  - Al entrar o salir de regiones #parallel, #critical u #ordered
  - Al salir de una región de worksharing (single, etc), a menos que se haya usado el #nowait
  - Durante cualquier set/unset de un lock
  - Inmediatamente antes e inmediatamente después de un momento de calendarización de tasks

#pragma omp taskwait

- Similar a barrier, pero para el uso de constructos task
- Así como *barrier* aplica a un equipo de threads, *taskwait* aplica a un grupo de tareas
- Hablaremos de Tasks a continuación...

Paralelismo de Tareas



Partición de tareas (trabajo)

- La directiva parallel for nos ayuda a crear partición de dominio (datos) en el programa.
  - Dependencias de datos (!)
- En otras ocasiones necesitamos particionar por tareas.
  - Concurrentes
  - No vinculadas necesariamente a un set de datos.
- Directiva *parallel sections*.

Directiva parallel sections

Nos permite paralelizar programas donde las tareas pueden ser concurrentes. (El orden de ejecución no es importante y puede ser no determinístico)

```
#pragma omp parallel
{
    ...
    #pragma omp sections
    ...
}
```

```
#pragma omp parallel sections
{
    ...
}
```

Directiva sections / section

- Podemos colocar dentro de un bloque sección, las tareas independientes que se ejecutan de forma concurrente
- *sections* puede tener cláusulas shared, private, num_threads...
- *section* no tiene cláusulas
- Si hay más hilos que section, entonces algunos estarán ociosos.

```
#pragma omp parallel sections
{
#pragma omp section
      {
      // tarea concurrente 0
      }
      …
#pragma omp section
      {
      // tarea concurrente M - 1
      }
}
```

BARRERA IMPLÍCITA

Directiva task

- OpenMP 3.0. Existían antes pero ahora podemos manejarlas de forma explícita. Tiene los siguientes elementos:
  - Bloque de código
  - Datos y variables asociados a la tarea (locales)
  - Referencia de hilos (opcional) que ejecutan la tarea.
- Crea el paquete (estructura) que describe la tarea y asigna ejecución a un hilo
- Desacopla las tareas (contrario a *section*). Puede ejecutarlas de forma dinámica y asíncrona (sin barreras explícitas)

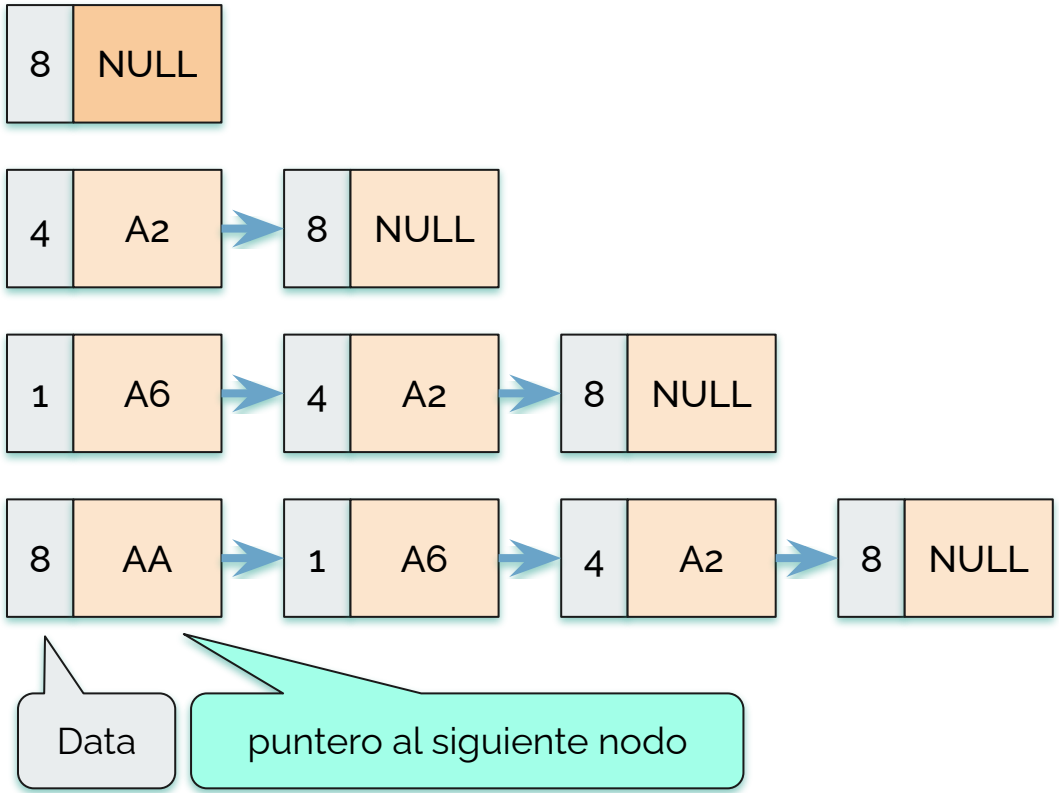
Listas vinculadas

Arrays cuando sabemos la cantidad de elementos.

Cualquier elemento

Linked lists cuando no tenemos certeza.

Secuencial



Recorrido y procesamiento de linked lists

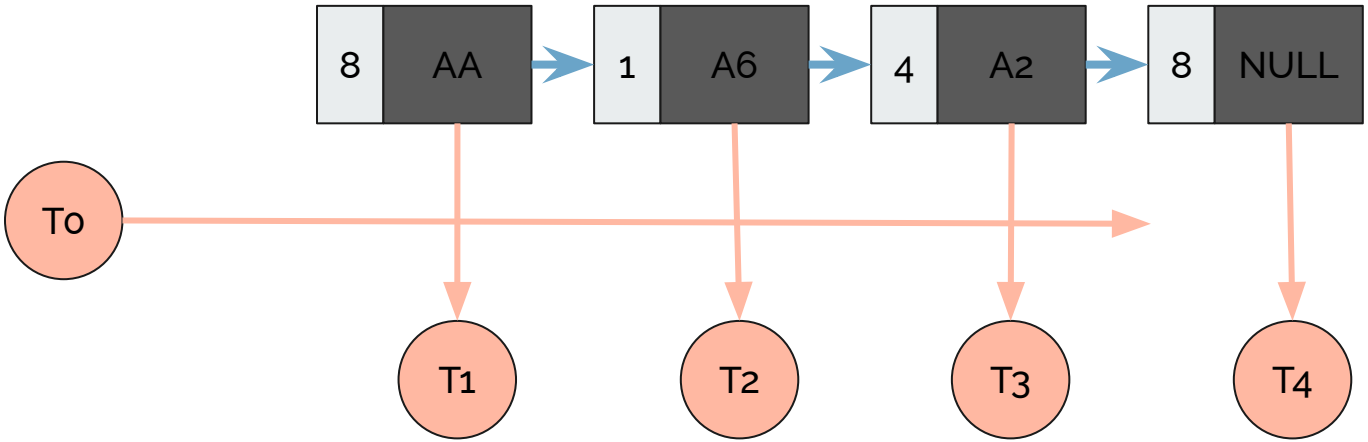
Un solo hilo recorre la lista (**pragma omp single**) mientras no se llegue al puntero NULL.

OpenMP crea hilos que realizan tareas concurrentes de procesar el dato del nodo correspondiente

```
template <class T>
struct Node
{
  T info;
  Node *next;
};
```

```
#pragma omp parallel
{
#pragma omp single
   {
    Node<int> *tmp = head;
    while (tmp != NULL)
        {
#pragma omp task
        process (tmp);
        tmp = tmp->next;
        }
    }
}
```

Recorrido y procesamiento de linked lists



Cláusulas de directiva task

Además de las cláusulas de cambio de scope:

- **if (expresión escalar)** – si evalúa la expresión a 0, la tarea se vuelve undeferred. Ello causa que esa tarea no se "pueda correr después", obligando a ejecutarla en ese momento. La tarea puede ser ejecutada por otro hilo. Si la tarea es ejecutada por el hilo que la inicia se conoce como included.
- **final (expresión escalar)** – cuando evalúa a TRUE, la tarea y sus derivadas se convierten en final e included. En otras palabras obliga a que se ejecute inmediatamente y por el mismo hilo

Cláusulas de directiva task

Además de las cláusulas de cambio de scope:

- **untied** – por defecto, una tarea está atada a un hilo. Si la tarea es suspendida, esperará hasta que el hilo que la inició la vuelva a tomar. En cambio, una tarea *untied* puede ser reiniciada por cualquier hilo.
- **mergeable** – una tarea fusionada (merged) comparte su ámbito de datos con la tarea que la generó. Esto permite a OpenMP decidir si genera una tarea merged al momento de que se genere una tarea undeferred (con *if* o *final*).

Ejercicio Individual (Corto 5)

(Entrega: miércoles 29 19:00)

- (1.25 pts) Refiera a sus códigos que han realizado anteriormente (i.e.: riemann, pi) y modifiquelo para utilizar constructos estudiados en esta presentación (atomic, sections, tasks, single/master, locks, etc). Mida tiempos de ejecución de ambas versiones del código y comparelos. Calcule el speedup y eficiencia de ambas versiones.
- (1.25 pts) Realice una versión secuencial de fibonacci recursivo. Utilice OpenMP para paralelizar, aplicando el uso de tasks. Apóyese de cláusulas *if* o *final* para evitar generar muchas tareas dinámicamente (experimente que pasa si no lo hace). Calcule el speedup y eficiencia de su programa paralelo. *(tip: ver Barlas 4.5.2, especialmente Listing 4.20)*

Referencias

1. **Barlas. G**. “Chapter 4 – Shared-memory programing: OpenMP”. *Multicore and GPU Programming – An Integrated Approach*. Morgan-Kaufmann. 2015.
2. **Pacheco, P**. “5. Shared-Memory Programming with OpenMP” *An Introduction to Parallel Programming*. Morgan-Kaufmann. 2011.
3. **Trobec, R. et al**. “3. Programming Multi-core and Shared Memory Multiprocessors Using OpenMP” *Introduction to Parallel Computing – From Algorithms to Programming on State-of-the-Art Platforms*. Springer. 2018.
4. **Rauber, T. Rünger, G**. “6.3 OpenMP” Parallel Programming for Multicore and Cluster Systems. Springer. 2010.