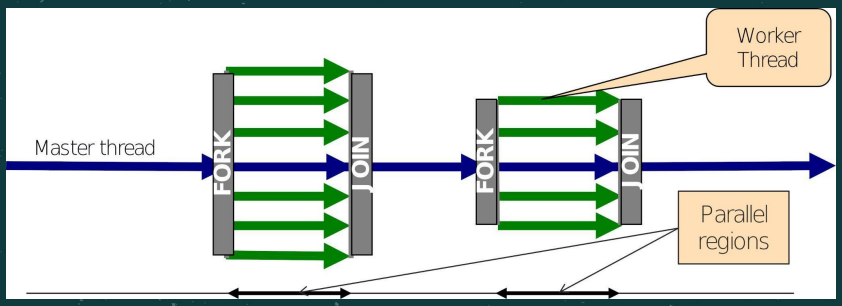

OpenMP
Fundamentos

OpenMP (Open Multi-Processing)

- Una extensión al lenguaje con constructores para programación paralela
  - Secciones críticas, acceso atómico, variables privadas, barreras
- Paralelización es ortogonal a la funcionalidad
  - Si el compilador no reconoce las directivas de OpenMP, el código sigue siendo funcional (aunque single-thread)
- Estándar en la industria: soportado por Intel, Microsoft, IBM, HP

OpenMP: Modelo de Ejecución

Modelo Fork-Join: el subproceso maestro genera un equipo de subprocesos según sea necesario



OpenMP: Modelo de Memoria

- Modelo de memoria compartida:
  - Los threads se comunican accediendo a variables compartidas.
- El compartir se define sintácticamente:
  - Cualquier variable que sea vista por dos o más threads es compartida
  - Cualquier variable que es vista por un solo thread es privada
- Posibilidad de problema de Race conditions:
  - Utilizar sincronización para protegerse de los conflictos
  - Cambiar cómo se almacenan los datos para minimizar la sincronización

OpenMP: entorno de datos

- Las variables globales son compartidas por los threads
- Variables privadas:
  - Variables de iteración en loops for
  - Variables que son declaradas localmente en una región paralela
  - Variables explícitamente declaradas como privadas (`#pragma omp parallel private(x)`)

OpenMP: Ejemplo de trabajo compartido

El `#pragma omp parallel` se usa para crear threads adicionales para llevar a cabo el trabajo encerrado por la construcción en paralelo. El thread original se lo denota como master thread con id de thread 0.

```
#include <omp.h>
…
#pragma omp parallel
{
  int id = omp_get_thread_num();
  int nt = omp_get_num_threads();
  printf("id: %d nt: %d\n", id ,nt);
}
```

La directiva #pragma es el método especificado por el estándar C para proporcionar información adicional al compilador, más allá de lo que se transmite en

Constructores de trabajo compartido

Se utiliza para especificar cómo asignar trabajo independiente a uno o a todos los threads.

- `omp for` u `omp do`: se utiliza para dividir las iteraciones de bucle entre los threads, también llamadas *construcciones de bucle*.
- `sections`: asignación de bloques de código consecutivos pero independientes a diferentes threads.
- `single`: especifica un bloque de código que es ejecutado por un solo thread, una barrera está implícita al final.
- `master`: similar a `single`, pero el bloque de código será ejecutado solo por el master thread y no implicará ninguna barrera al final.

OpenMP: Ejemplo de trabajo compartido

```
answer1 = long_computation_1();
answer2 = long_computation_2();

if (answer1 != answer2) { … }
```

Podemos paralelizarlo:

```
#pragma omp sections
{
  #pragma omp section
  answer1 = long_computation_1();
  #pragma omp section
  answer2 = long_computation_2();
}
```

OpenMP: Ejemplo `#parallel for`

```
int a[100000];
#pragma omp parallel
{
  #pragma omp for
  for (int i = 0; i < 100000; i++)
  {
    a[i] = 2 * i;
  }
}
```

Este ejemplo es vergonzosamente paralelo ya que depende únicamente del valor de i. La combinación de banderas `parallel for` de OpenMP hace que se divida esta tarea entre sus threads de trabajo. Cada thread recibirá una versión única y privada de la variable. Por ejemplo, con dos threads de trabajo, un thread puede recibir una versión de i que se ejecuta de 0 a 49999 mientras que el segundo obtiene una versión que se ejecuta de 50000 a 99999.

Desafíos de `#parallel for`

- Balanceo de carga
  - Si todas las iteraciones se ejecutan a la misma velocidad, los procesadores se utilizan de manera óptima
  - Si algunas iteraciones son más rápidas, algunos procesadores pueden quedar inactivos, lo que reduce la velocidad
  - No siempre conocemos la distribución del trabajo, es posible que debamos redistribuirlo dinámicamente
- Granularidad
  - La creación y sincronización de threads lleva tiempo
  - La asignación de trabajo a threads en un `#parallel for` puede llevar más tiempo que la ejecución en sí
  - Necesidad de juntar el trabajo en chunks grandes para superar la sobrecarga del threading

Trade-off entre balanceo de carga y granularidad del paralelismo

Cláusulas de atributos de Data sharing

- `shared`: los datos declarados fuera de una región paralela se comparten, lo que significa que todos los threads pueden verlos y accederlos simultáneamente. Por defecto, se comparten todas las variables de la región de trabajo compartido, excepto el contador de iteraciones del bucle.
- `private`: los datos declarados dentro de una región paralela son privados para cada thread, lo que significa que cada thread tendrá una copia local y la usará como una variable temporal. Una variable privada no se inicializa y su valor no se mantiene fuera de la región paralela. Por defecto, los contadores de iteraciones en las construcciones de bucle de OpenMP son privados.
- `default`: permite al programador establecer que el scope por defecto de los datos dentro de una región paralela será `shared` o `none`. La opción `none` obliga al programador a declarar cada variable en la región paralela utilizando las cláusulas de atributo de intercambio de datos.

Cláusula: private

```
int i = 10;
#pragma omp parallel private(i)
{
  // variable i is not initialized
  printf("thread %d: i = %d\n", omp_get_thread_num(), i);
  i = 1000;
}
printf("private i = %d\n", i);
```

Cláusulas de atributos de Data sharing (cont.)

- `firstprivate`: como `private`, excepto que es inicializado al valor original.
- `lastprivate`: como `private`, excepto que el valor de estos datos privados se copiará en una variable global con el mismo nombre fuera de la región paralela si la iteración actual es la última iteración en el bucle paralelizado.
- `reduction`: permite realizar reducciones del estilo `reduction_variable = reduction_variable operator value` en un bucle paralelizado de manera segura (evitando race conditions). Eslo lo logra haciendo que cada thread inicialice y actualice la `reduction_variable` de manera independiente para luego combinar los resultados de cada thread en una variable global.

Cláusulas: firstprivate/lastprivate

```
int i = 10;
#pragma omp parallel private(i)
{
  // variable i is not initialized
  printf("thread %d: i = %d\n", omp_get_thread_num(), i);
  i = 1000;
}
printf("private i = %d\n", i);

#pragma omp parallel firstprivate(i)
{
  // variable i is initialized to its original value
  printf("thread %d: i = %d\n", omp_get_thread_num(), i);
  i = 1000;
}
printf("firstprivate i = %d\n", i);
```

```
#pragma omp parallel
{
   #pragma omp for lastprivate(i)
      for (i=0; i<n-1; i++)
         a[i] = b[i] + b[i+1];
}
a[i]=b[i]; // i = n-1
```

Cláusula:

`reduction(operator: reduction_var)`

```
int sum = 0;
int val = 1;

#pragma omp parallel for reduction(+: sum)
for (int i = 0; i < 100; i++)
{
  sum = sum + val;
}

printf("sum: %d\n", sum);
```

Hay un race condition si reduction(+: sum) clause no es incluida

Cláusulas de sincronización

- `critical`: el bloque de código dentro del scope de esta cláusula será ejecutado por un solo thread a la vez, y no ejecutado simultáneamente por varios threads. A menudo se usa para proteger los datos compartidos de las condiciones de carrera.
- `atomic`: la actualización de memoria (write o read-modify-write) en la siguiente instrucción se realizará de forma atómica. `atomic`, a diferencia de `critical`, solo puede proteger a una sola asignación y solo puede usarse con operadores específicos.

Cláusula: `crítical [(name)]`

Modelo de cola en el que una tarea se quita de la cola y se trabaja en ella. Para evitar que muchos threads quiten la misma tarea, la operación de dequeue de la cola debe estar en una sección crítica. Debido a que las dos colas de este ejemplo son independientes, están protegidas por directivas `critical` con diferentes nombres, `critical_section1` y `critical_section2`. Si los nombres coinciden entre diferentes secciones entonces solo un thread puede entrar a una de las secciones.

```
#pragma omp parallel shared(x, y) private(x_next, y_next)
{
  #pragma omp critical ( critical_section1 )
  {
    x_next = dequeue(x);
  }
  work(x_next);
  #pragma omp critical ( critical_section2 )
  {
    y_next = dequeue(y);
  }
  work(y_next);
}
```

Cláusulas de sincronización (cont.)

- `ordered`: fuerza que un bucle `for` dentro de una región paralelizada se ejecute de manera secuencial (sin paralelizar).
- `barrier`: cada thread espera hasta que todos los demás threads de un equipo hayan llegado a este punto. Una construcción de work-sharing tiene una sincronización de barrera implícita al final.
- `nowait`: especifica que los threads que completan su trabajo pueden continuar sin esperar a que finalicen todos los threads del equipo. En ausencia de esta cláusula, los threads encuentran una barrera de sincronización al final de la construcción de trabajo compartido.

Cláusula: nowait

```
#pragma omp parallel
{
  #pragma omp for nowait
  for (i=1; i<n; i++)
    b[i] = (a[i] + a[i-1]) / 2.0;
  #pragma omp for nowait
  for (i=0; i<m; i++)
    y[i] = sqrt(z[i]);
}
```

Cláusula de control `if`

La cláusula `if` hará que los threads paralelicen la tarea solo si se cumple una condición. De lo contrario, el bloque de código se ejecuta en serie.

Ejemplo: `#pragma omp parallel if (n>100000)` corre el código delimitado por esta cláusula solo cuando `n > 10000`.