Programación paralela basada en tareas con OpenMP

Christian Asch

1. Introducción a la programación paralela basada en tareas

1.1. Qué es la programación paralela basada en tareas?

La programación paralela que hemos aprendido funciona bien para problemas regulares, donde el trabajo a realizar y los patrones de acceso de memoria se pueden mapear fácilmente a índices de uno o varios ciclos. Cuando los problemas son irregulares, las técnicas que hemos aprendido pueden llevar a desbalances de carga. Estos problemas irregulares incluyen por ejemplo operaciones con estructuras de datos dispersas o problemas con estructuras de control complejas.

1.2. Tareas vs hilos

Las tareas son un concepto más abstracto que los hilos. A nivel de implementación de OpenMP, las tareas se manejan con hilos sin embargo, al nivel de programación de aplicaciones, las tareas funcionan por sí mismas.

1.3. Problema motivador

Imaginemos que tenemos datos guardados en una lista enlazada y debemos de aplicar una operación a cada elemento de la lista. No sabemos el tamaño de la lista y no sabemos cuánto dura cada operación. Dada esta situación no podemos fácilmente paralelizar el programa con lo que sabemos al momento. Una forma sería contar la cantidad de elementos en la lista, copiarlos a un arreglo y después utilizar un `#pragma omp parallel for`, sin embargo esta no siempre es una opción práctica.

```
 1  #include <omp.h>
 2  #include <stdlib.h>
 3  #include <stdio.h>
 4
 5  typedef struct Node {
 6    int data;
 7    int result;
 8    struct Node *next;
 9  } Node;
10
11  int es_primo(int num)
12  {
13    if(num <= 2)
14    {
15      return 1;
16    }
17    for(int i = 2; i < num; ++i)
18    {
19      if(num % i == 0)
20      {
21        return 0;
22      }
```

```
23    }
24    return 1;
25  }
26
27  void procesar_lista(Node *node)
28  {
29    if(node != NULL)
30    {
31      node->result = es_primo(node->data);
32      procesar_lista(node->next);
33    }
34  }
35
36  void imprimir_lista(Node *node)
37  {
38    if(node != NULL)
39    {
40      printf("(%d,%d)\n", node->data, node->result);
41      imprimir_lista(node->next);
42    }
43
44  }
45
46  void inicializar_lista(Node **node, int start, int finish)
47  {
48    (*node) = malloc(sizeof(Node));
49    (*node)->data = start;
50    (*node)->result = 0;
51    if(finish != start)
52    {
53      inicializar_lista(&(*node)->next, start+1, finish);
54    }
55    else {
56      (*node)->next = NULL;
57    }
58  }
59
60  void borrar_lista(Node **node)
61  {
62    if(*node)
63    {
64      Node *next = (*node)->next;
65      (*node) = NULL;
66      borrar_lista(&next);
67    }
68  }
69
70  int main()
71  {
72    Node * node = malloc(sizeof(Node));
73    int size = 100000;
74    int start = 10000000;
75    inicializar_lista(&node, start, start+size);
```

```
76    procesar_lista(node);
77    //imprimir_lista(node);
78    borrar_lista(&node);
79    return 0;
80  }
```

2. Programación paralela basada en tareas básica con OpenMP

2.1. Creación de tareas en OpenMP

El constructo `task` crea una tarea de OpenMP. Cada tarea es una unidad independiente de trabajo con dos componentes:

- El código que va a ejecutar, incluyendo cualquier función que encuentre (la región de la tarea).
- El ambiente de datos asociado.

2.2. El constructo `task`

```
#pragma omp task [cláusulas]
```

Cuando un hilo se encuentra a este constructo tiene dos opciones, puede empezar a ejecutarlo en ese mismo momento o podría deferir la ejecución a otro momento. Esta segunda posibilidad es lo que permite que las tareas sean ventajosas para el problema de desbalance de cargas.

2.3. Fibonacci

La serie de Fibonacci se puede definir por la siguiente relación de recurrencia:
$F_0 = 0,\ F_1 = 1$
$F_n = F_{n-1}\ + F_{n-2}$
Una versión serial del programa es la siguiente:

```
 1  #include <stdio.h>
 2
 3  unsigned int fib(unsigned int n)
 4  {
 5    unsigned int x,y;
 6    if (n < 2) return n;
 7    x = fib(n-1);
 8    y = fib(n-2);
 9    return x + y;
10  }
11
12  int main()
13  {
14    unsigned int number = 30;
15    unsigned int result = fib(number);
16    printf("El número de Fibonacci %u es %u\n", number, result);
17    return 0;
18  }
```

Esta no es la mejor forma de codificar este algoritmo, sin embargo es útil para explicar el tema.

Para paralelizar este programa utilizando tareas vemos que este programa genera un árbol binario de llamadas a funciones y que cada llamado depende de aquellos que están más abajo en el árbol. Podemos traducir cada llamado a función como una tarea.

```
 1  #include <stdio.h>
 2
 3  unsigned int fib(unsigned int n)
 4  {
 5    unsigned int x,y;
 6    if (n < 2) return n;
 7  #pragma omp task shared(x)
 8    x = fib(n-1);
 9  #pragma omp task shared(y)
10    y = fib(n-2);
11  #pragma omp taskwait
12    return x + y;
13  }
14
15  int main()
16  {
17    unsigned int number = 30;
18    unsigned int result = 0;
19  #pragma omp parallel
20    {
21  #pragma omp single
22      result = fib(number);
23    }
24    printf("El número de Fibonacci %u es %u\n", number, result);
25    return 0;
26  }
```

Aquí podemos ver varios constructos nuevos:

```
#pragma omp taskwait
```

Este constructo le indica al hilo que espere a que sus tareas hijas terminen antes de continuar ejecutando. En este caso es necesario ya que necesitamos el valor de `x` y `y` para poder calcular su suma.

```
#pragma omp single
```

Este constructo hace que el bloque interno sólo sea ejecutado por un hilo, este hilo es el que crea las tareas y espera mientras los otros hilos las ejecutan.

2.4. Patrón de Divide and Conquer

2.5. Práctica:

Utilice el patrón de Divide and Conquer para escribir un programa que calcule Pi con tareas.

```
#include <stdio.h>
#include <omp.h>

int main()
{
  const int num_steps = 1024 * 1024 * 1024;
  double x, pi, sum = 0.0;
  double start_time, end_time;

  double step = 1.0/(double) num_steps;
```

```
  start_time = omp_get_wtime();

  for(int i = 0; i < num_steps; ++i)
  {
    x = (i + .5) * step;
    sum += 4./(1. + x * x);
  }

  pi = step * sum;
  end_time = omp_get_wtime() - start_time;

  printf("pi=%f, %d steps, %f seconds", pi, num_steps, end_time);

  return 0;
}

#include <stdio.h>
#include <omp.h>

#define MIN_BLK 1024 * 256

int pi_component(int start, int finish, double step)
{
  double x, sum1, sum2, sum = 0.0;
  if(finish - start < MIN_BLK)
  {
    for(int i = start; i < finish; ++i)
      {
        x = (i + .5) * step;
        sum += 4./(1. + x * x);
      }
  }
  else
  {
    int iblk = finish - start;
    #pragma omp task shared(sum1)
    sum1 = pi_component(start, finish - iblk/2, step);
    #pragma omp task shared(sum2)
    sum2 = pi_component(finish - iblk/2, finish, step);
    #pragma omp taskwait
    sum = sum1 + sum2;
  }
  return sum;
}

int main()
{
  const int num_steps = 1024 * 1024 * 512;
  double step = 1./(double) num_steps;

  double start_time = omp_get_wtime();
  double sum = 0.0;
  #pragma omp parallel
  {
```

```
    #pragma omp single
    sum = pi_component(0, num_steps, step);
  }
  double pi = step * sum;
  double end_time = omp_get_wtime() - start_time;

  printf("pi=%f, %d steps, %f seconds", pi, num_steps, end_time);
}
```

3. Resumen de OpenMP

3.1. Threads

`#pragma omp parallel`

Crea la región paralela. El número de hilos solicitados se puede controlar con la variable de entorno `OMP_NUM_THREADS` o con `omp_set_num_threads()`, sin embargo el número de hilos que se asignan depende del runtime.

3.2. Worksharing

`#pragma omp for`

Distribuye los hilos de tal forma que cada uno trabaja en regiones del ciclo for. La distribución de trabajo se puede controlar con `schedule`.

`#pragma omp single`

Hace que sólo un hilo ejecute el código mientras los otros hilos esperan. Es útil para regiones de código que sólo se pueden o deben de ejecutar por un sólo hilo.

3.3. Tareas

`#pragma omp single`

3.4. Barreras

`#pragma omp barrier`

`#pragma omp critical`

3.5. Ambiente de datos

```
shared
private
firstprivate
```

3.6. Reducciones

`reduction(operador:variables)`

3.7. Variables de entorno y funciones del runtime

```
OMP_NUM_THREADS
void omp_set_num_threads(int)
int omp_get_num_threads()
int omp_get_thread_num()
double omp_get_wtime()
```