Patrones y esquemas de programación paralela

Jesús Sánchez Cuadrado

Resumen. En este documento se presentan esquemas de programación paralela.

Índice

1. Introducción

Un patrón de diseño es una *una solución a un tipo recurrente de problemas en determinado contexto*. Los patrones de diseño codifican la experiencia acumulada, los diseños y las mejores prácticas para resolver cierto tipo de problemas en un dominio dado.

En [2] se define un lenguaje de patrones, OPL, que tiene como objetivo ayudar al diseño de sistemas y algoritmos paralelos. En OPL los patrones se organizan de manera jerarquica según en qué del sistema estén centrados. La figura 1 muestra la organización de los patrones.

Our Pattern Language (OPL)

Es un esfuerzo para documentar patrones utilizados en el ámbito de la computación paralela. Está promovido por Kurt Keutzer (Berkeley) y Tim Mattson (Intel).
Se puede consultar en `https://patterns.eecs.berkeley.edu`

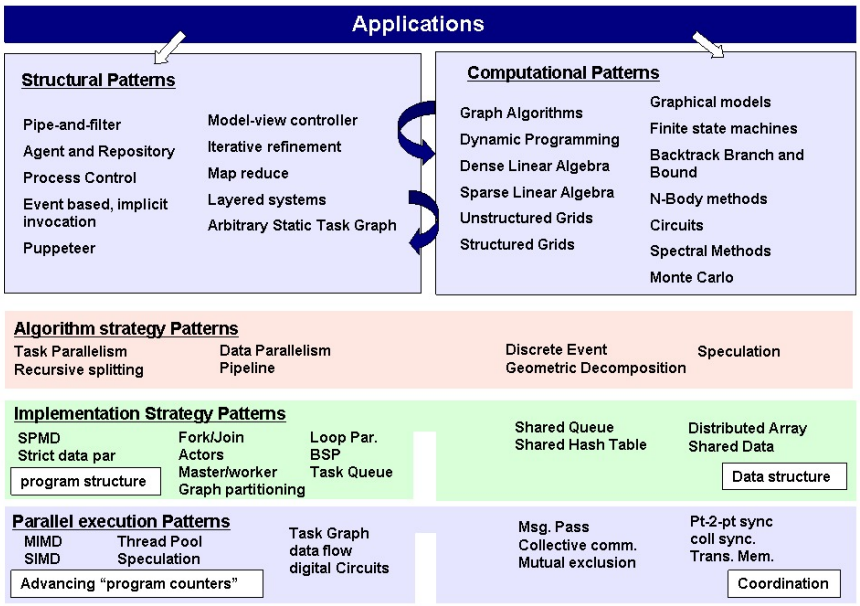

Figura 1: Organización de los patrones en OPL. Tomada de `https://patterns.eecs.berkeley.edu/?page_id=98`

- Patrones estructurales. Describen cómo se organiza la aplicación y de qué manera interactuan sus componentes.
- Patrones computacionales. Describen diferentes tipos de computaciones, como por ejemplo, algoritmos sobre grafos. Es decir, son patrones que se aplican a tipos de problemas específicos o dominios específicos.
- Patrones algorítmicos. Estos patrones describen cómo extraer paralelismo para cierto tipo de algoritmos.
- Estrategias de implementación. Las estructuras o construcciones que ofrecen los lenguajes o librerías de programación paralela para implementar los patrones.
- Patrones de ejecución. Se refieren a los mecanismos de bajo nivel para poder ejecutar el código en paralelo.

En este documento nos centraremos principalmente en los patrones algorítmicos (esquemas paralelos en [1]).

2. Diseño de algoritmo paralelos

El diseño de un algoritmo paralelo implica especificar qué pasos se seguirán (la secuencia de instrucciones, como en un algoritmo secuencial), pero también decidir qué partes de van a ejecutar en paralelo y cómo se van a organizar los datos.

- Identificar las partes que se pueden ejecutar concurrentemente.
- Mapear las partes concurrentes a procesos que se ejecutan en paralelo.
- Distribuir los datos de entrada entre los procesos, así como decidir cómo se distribuyen las salidas del programa. Se utilizará la palabra proceso para designar un elemento que puede realizar cálculos en paralelo. En el caso de OpenMP y Java un proceso será equivalente a un hilo.
- Gestionar los datos compartidos entre varios procesos.
- Sincronizar los procesos en diferentes etapas de la ejecución.

En general, dependiendo del problema que se esté abordando y del algoritmo que se diseñe se utilizarán unas u otras estrategias para abordar estas cuestiones. Estas decisiones implicarán que la solución tendrá ciertas características de rendimiento y de consumo de recursos, y que posiblemente funcione mejor en determinado tipo de arquitecturas y modelos de programación.

3. Paralelismo y particionado de datos

3.1. Problema

Los datos de entrada se van a tratar de igual manera. En muchas ocasiones son algoritmos númericos que hay que recorrer y realizar alguna operación más o menos costosa sobre ellos.

3.2. Contexto

Este patrón es aplicable cuando los procesos tratan de igual manera todos los datos. Es bastante utilizado cuando se trabaja con grandes volumenes que datos que se encuentran almacenados en vectores o matrices. Cuando se realizan cálculos matemáticos, éstos suelen ser bastante regulares y facilitan la aplicación del patrón.

3.3. Solución

Es necesario establecer una estrategia para dividir el trabajo entre los diferentes procesos, es decir, realizar el *particionado de datos*. Esta tarea es más sencilla cuando todos los datos se tratan de igual manera y por tanto el tiempo de ejecución en procesar cada partición es similar en cada proceso.

Se puede utilizar paralelismo implícito o explícito, dependiendo del soporte ofrecido por la tecnología paralela utilizada.

- Implícito: El sistema automáticamente realizará la partición del trabajo, posiblemente a través de alguna directiva (como en OpenMP).
- Explícito: El programador debe calcular el qué bloques de datos se asignan a cada proceso. En el caso de memoria compartida se calcularán las posiciones de memoria que debe tratar cada hilo.

  En un sistema de paso de mensajes será necesario construir los mensajes con los datos explícitamente. Además, en este caso hay comunicaciones

implicadas por lo que ese coste debe tenerse en cuenta. Normalmente el volumen de computación tiene que ser un orden mayor (10 veces mayor) que el de comunicaciones. Por ejemplo, en una multiplicación matrix por vector el coste computacional es n^2 por lo que que si hay que distribuir la matriz completa (con coste de comunicaciones n^2), entonces no compensa utilizar paso de mensajes. Un alternativa sería idear una estrategia para que cada proceso pudiera cargar su parte de la matriz de manera local. Otro ejemplo es la multiplicación de dos matrices, donde el coste de computaciónes n^3 por lo que sí compensa utilizar paso de mensajes.

3.4. Ejemplo: Suma de n números

Un ejemplo simple de paralelismo de datos es la suma de n números. Es posible realizar una partición del trabajo, de manera que cada proceso se encarga de un grupo de números. Todos los procesos realizan la misma operación. En este caso, cada proceso obtiene un resultado parcial que hay que agregar.

La figura 2 muestra un ejemplo de cómo se distribuye el trabajo.

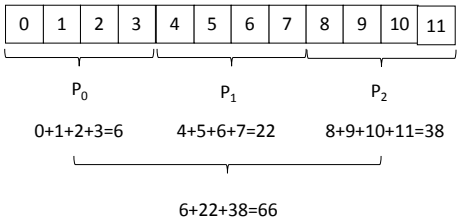

Figura 2: Partición de datos para la suma de n números.

3.4.1. OpenMP

En OpenMP el paralelismo de datos es normalmente muy sencillo de implementar, puesto que la directiva `parallel for` está especialmente diseñada para ello y realiza la división del trabajo de manera automática. Además, se puede usar la clausula `schedule` para seleccionar la estrategia de división del trabajo (por bloques contiguos, cíclicamente, etc.). El siguiente listado muestra su uso.

```
#pragma omp parallel for reduction(+:suma)
for(int i = 0; i < N; i++) {
  suma += datos[i];
}
```

En memoria compartida dividir el trabajo suele ser más simple porque los hilos acceden directamente a las posiciones de memoria que les corresponden. Por tanto, no es necesario "partir" en arrays diferentes los datos.

3.4.2. MPI

En MPI es necesario distribuir físicamente los datos que debe traer cada proceso. Por tanto, el particionado de datos debe ser explícito. El siguiente código muestra la implementación en MPI, utilizando `MPI_Scatter` para dividir el trabajo y `MPI_Reduce` para recolectar los datos en el maestro, realizando la agregación de los resultados parciales (sumándolos).

```
#include <stdio.h>
#include <stdlib.h>
#include <mpi.h>

#define ARRAYSIZE 100

int main ( int argc, char *argv[] )
{
  int rank, world_size;
  int *data;

  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  MPI_Comm_size(MPI_COMM_WORLD, &world_size);

  /* El maestro es el encargado de cargar los datos */
  if(rank == 0) {
     data = (int *) calloc(ARRAYSIZE, sizeof(int));
     for (int i = 0; i < ARRAYSIZE; i++)
  data[i] = i + 1;
```

```
  }

  /* Dividir el trabajo */
  int chunksize = (ARRAYSIZE / world_size);
  int leftover = (ARRAYSIZE % world_size);

  int *tosum = (int *) malloc(chunksize * sizeof(int));

  /* Enviar el trozo saltandose la parte sobrante que le toca al maestro */
  MPI_Scatter(&data[leftover], chunksize, MPI_INT,
        tosum, chunksize, MPI_INT, 0, MPI_COMM_WORLD);

  int local_sum = 0;

  /* El maestro suma la parte que le toca solo a el */
  if (rank == 0) {
    for(int i = 0; i < leftover; i++)
      local_sum += data[i];
  }

  /* Todos los nodos suman su parte */
  for(int i = 0; i < chunksize; i++)
    local_sum += tosum[i];

  int sum;
  MPI_Reduce(&local_sum, &sum, 1, MPI_INT, MPI_SUM,0, MPI_COMM_WORLD);

  /* El maestro recibe los resultados parciales y los suma */
  if(rank == 0)
  {
     printf("Resultado %d\n", sum);
     free(data);
  }

  free(tosum);

  MPI_Finalize();
  return 0;
}
```

3.4.3. Java

Ejercicio: Implementar en Java

3.5. Ordenación por rango

El algoritmo de ordenación por rango se basa en contar cuántos elementos hay menores (o iguales) que cada uno de los elementos array, esto es, su rango. El rango nos dice la posición que ocupa el elemento ordenado. Así, si el elemento que originalmente estaba en la posición 7 tiene 2 elementos menores, su posición en el nuevo array ordenado es la 2. El siguiente código muestra la implementación de este algoritmo.

```
for(i=0; i<n; i++)
  for(j=0;j<n;j++)
    if (a[i]>a[j] || ((a[i] == a[j]) && (i > j)))
      r[i]+=1;

for(i = 0; i < n; i++)
  b[r[i]] = a[i];
```

Aunque este algoritmo tiene coste n^2+n, su paralelización es sencilla porque es posible dividir el trabajo en grupos que se procesan independientemente.

3.5.1. OpenMP

La paralelización en OpenMP es directa. En el primer bucle no hace falta ninguna sección crítica porque cada hilo utiliza un i diferente para escribir `r[i]`. En el segundo bucle tampoco hace falta porque se cumple que los valores de `r[i]` son todos diferentes (esto es, son los índices del array) y por tanto el acceso a `b[r[i]]` es siempre disjunto.

```
#pragma omp parallel for private(i, j)
for(i=0; i<n; i++)
  for(j=0;j<n;j++)
    if (a[i]>a[j] || ((a[i] == a[j]) && (i > j)))
      r[i]+=1;

#pragma omp parallel for private(i)
for(i = 0; i < n; i++)
  b[r[i]] = a[i];
```

Es posible mejorar un poco el algoritmo fusionando ambos bucles. Esto es posible porque una vez que r[i] se ha calculado (se ha obtenido el rango para el elemento en el índice i, ya es posible copiarlo al array resultado `b`.

```
#pragma omp parallel for private(i, j)
for(i=0; i<n; i++)
  for(j=0;j<n;j++)
    if (a[i]>a[j] || ((a[i] == a[j]) && (i > j)))
      r[i]+=1;

  /* En este punto r[i] ya ha sido calculado */
  b[r[i]] = a[i];
```

4. Paralelismo de tareas

4.1. Problema

Este patrón aborda la cuestión de *cómo descomponer un problema en tareas que se puedan ejecutar de manera concurrente*. Es aplicable cuando se tiene un algoritmo que se puede describir como un conjunto de tareas que trabajan sobre un conjunto de datos. Se desea abordar el problema de cómo ejecutar estas tareas de manera paralela de una forma eficiente.

4.2. Contexto

La descomposición de un problema para que pueda ser paralelizado es esencial en la programación paralela. Siempre hay dos cuestiones a tener en cuenta en esta descomposición: qué tareas se ejecutan de manera concurrente y sobre qué datos.

Cuando el problema puede dividirse de manera natural en tareas independientes (o casi independientes) suele ser más sencillo orientar la solución hacia una descomposición basada en tareas. En este caso la distribución y el particionado de datos tiene menos importancia que en el paralelismo de datos.

4.3. Solución

La solución implica decidir qué partes de la computación se convierten en tareas y cómo organizar su ejecución (*scheduling*). La solución suele estar relacionada con la decomposición funcional del problema secuencial, esto es, qué cálculos se realizan sobre los datos y de qué forma se pueden descomponer estos cálculos para convertirlas en tareas (idealmente independientes entre sí).

Hay varios compromisos que tener en cuenta:

- **Granularidad de las tareas**. El problema a resolver se divide en un cierto número de tareas a resolver. El objetivo es tener un número grande tareas pequeñas para que se pueda balancear la carga, pero a medida que aumenta el número de tareas hay una mayor sobrecarga a la hora de gestionarlas. Es importante encontrar un buen equilibrio entre estos dos aspectos: número de tareas para balancear la carga y sobrecarga.
- **Interacción entre las tareas**. En ciertos algoritmos las tareas proceden de manera casi independiente, y en otros es necesario realizar más trabajo de sincronización. Suele ser buena idea intentar minimizar la cantidad de sincronización, a veces incluso a costa de duplicar ciertos cálculos.

4.4. Ejemplo: El conjunto de Mandelbrot

El conjunto de Mandelbrot es un tipo de fractal que se construye coloreando cada pixel de acuerdo a la siguiente ecuación:

$$Z_{n+1} = Z_n^2 + C \tag{1}$$

donde Z y C son números complejos. Dada una imagen, el número complejo C representa un pixel (la parte imaginaria la coordenada y y la parte real la coordenada x). El algoritmo para calcular el conjunto de Mandelbrot consiste en aplicar la ecuación anterior para cada pixel, comenzando con $Z_0 = C$ y hasta un cierto límite n. Si el valor absoluto de Z no diverge después un cierto número de iteraciones entonces el pixel pertenece al conjunto y se pinta en negro. Si no, entonces se pinta de un color según cuánto ha divergido. El siguiente listado muestra el pseudocódigo.

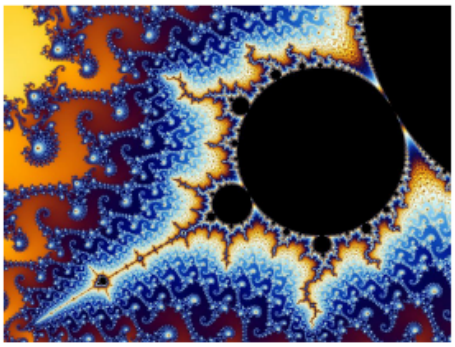

Figura 3: El conjunto de Mandelbrot.

```
float epsilon = 0.0001; // The step size across the X and Y axis
float x;
float y;

int maxIterations = 10;
int maxColors = 256; // Change as appropriate for your display.

Complex Z;
Complex C;
int iterations;
for(x=-2; x<=2; x+= epsilon) {
   for(y=-2; y<=2; y+= epsilon) {
      iterations = 0;
      C = new Complex(x, y);
      Z = new Complex(0,0);
      while(Complex.Abs(Z) < 2 && iterations < maxIterations) {
         Z = Z*Z + C;
         iterations++;
      }

      // Significa que no ha divergido (ha quedado por debajo de 2)
      if (iterations == maxIterations)
        dibujar(x, y, black);
      else
        dibujar(x, y, maxColors % iterations);
   }
}
```

Como puede observarse en este algoritmo cada pixel se calcula de manera totalmente independiente del resto. Si la función dibujar escribe en un buffer de una imagen, no es necesario utilizar ninguna sección crítica.

4.4.1. OpenMP

En OpenMP resulta natural utilizar secciones y tareas para abordar este tipo de problemas. También es posible utilizar paralelismo a nivel de bucle si las

tareas se generan de manera natural como índices.

Cuando hay una descomposición funcional clara y es posible asignar una o más funciones a un hilo las secciones resultan adecuadas y sencillas de utilizar. Las secciones permiten una división estática cuando las tareas son conocidas de antemano.

En el ejemplo del conjunto de Mandelbrot la solución más sencilla es utilizar un `parallel for`.

4.5. Discusión

Este patrón está relacionado con los algoritmos "embarransingly parallel", esto es, algoritmos para los que de forma natural se puede encontrar una gran concurrencia. En [1] a este tipo de algoritmos se les llama Algoritmos Relajados.

Referencias

[1] F. Almeida, D. Giménez, J. M. Mantas, and A. M. Vidal. *Introducción a la programación paralela*. Thompson Paraninfo, 2008.

[2] K. Keutzer and T. Mattson. Our pattern language (opl): A design pattern language for engineering (parallel) software. In *ParaPLoP Workshop on Parallel Programming Patterns*, volume 14, pages 10–1145, 2009.