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`

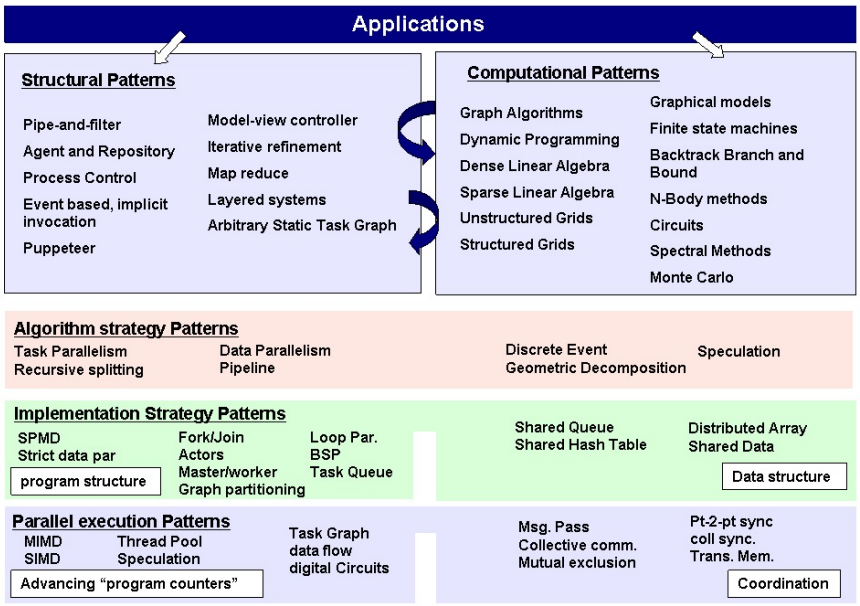


Figura 1: Organización de los patrones en OPL. Tomada de `https://patterns.eecs.berkeley.edu/?page_id=98`

- 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.

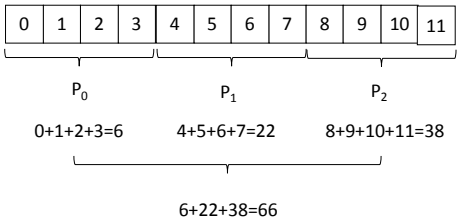


Figura 2: Partición de datos para la suma de n números.

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.

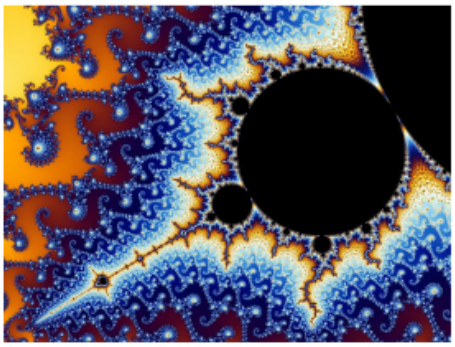


Figura 3: El conjunto de Mandelbrot.

```
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.

5. Descomposición geométrica

5.1. Problema

Este patrón se da cuando el problema que se desea resolver permite ser descompuesto en trozos, sobre los que se hace un cálculo, pero es necesario actualizar o intercambiar información entre los trozos vecinos.

En [1] a este patrón se le denomina *Paralelismo síncrono* porque lo habitual es que ese tipo de problemas se implemente como una serie de iteraciones, y en cada iteración los procesos deben intercambiar información entre ellos.

5.2. Contexto

En este tipo de problemas hay una estructura de datos principal, por ejemplo una matriz o un array, sobre la que se realizan operaciones de forma sucesiva. Al mismo tiempo, es posible descomponer la estructura de datos en trozos que se asignan a procesos. En cada paso de esta sucesión de operaciones, los procesos deben intercambiar información sobre sus trozos para que otros trozos puedan ser actualizados.

- Cada proceso realiza el mismo trabajo sobre una porción distinta de los datos.
- Al final de cada paso o iteración del algoritmo existe una sincronización entre los procesos, que puede ser local (p.ej., pasar información a los vecinos) o global (p.ej., pasar información a todos los otros procesos).
- El proceso suele finalizar cuando se han realizado un número fijo de pasos o iteraciones, o bien cuando se cumple algún criterio de convergencia.

5.3. Solución

Para resolver este tipo de problemas es necesario idear un forma de particionar los datos así como establecer la manera en que los procesos comunican sus resultados a otros procesos.

El rendimiento de la paralelización de este tipo de algoritmos depende de si hay comunicaciones o no. En memoria compartida el intercambio de información entre hilos suele ser más sencillo. En cambio, con paso de mensajes, es necesario enviar la información explícitamente en cada "paso de sincrononización".

5.4. Ejemplo: Difusión de calor en una placa

Dada una placa cuadrada con valores de temperatura fijos en los bordes, pero con valores en el interior que varían según la temperatura de sus vecinos, se desea calcular los valores de estos puntos en el interior una vez que se estabiliza.

La figura 4 muestra cómo se discretiza el problema para realizar la simulación. El problema que se desea resolver por tanto es, *dada una placa de metal para la que se conoce la temperatura en los bordes, que permanece fija, cuál es la temperatura de los puntos en el interior.*

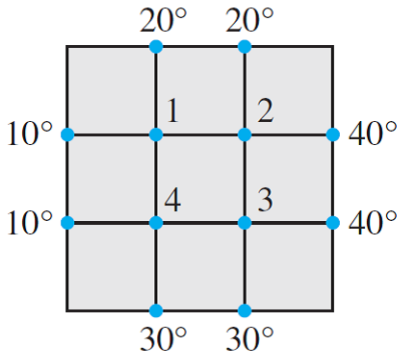


Figura 4: Discretización de una placa cuadrada con valores de temperatura.

Los valores de cada punto de la placa en un instante dado se calculan considerando los valores de sus cuatro vecinos en el instante anterior. Así, tenemos que el valor de un punto en instante t es $V^t(i,j)$[1]:

$$V^t(i,j) = \frac{V^{t-1}(i-1,j) + V^{t-1}(i+1,j) + V^{t-1}(i,j-1) + V^{t-1}(i,j+1)}{4} \tag{2}$$

El procedimiento de simulación consiste en establecer una estimación de los valores en el interior y calcular nuevos valores hasta que converjan. A este método que utiliza las estimaciones de la iteración i para calcular nuevas estimaciones en la iteración $i+1$ se le denomina método de Jacobi.

A continuación se muestra el algoritmo en C:

```
// Inicializar a y b
double a[][], b[][] = ...;

// Iterar hasta el maximo de iteraciones o hasta que converja
for(int it = 0; it < num_iteraciones; it++) {
  copiar b en a;

  // Actualizar cada celda en funcion de sus vecinos
  for(i = 1; i < n - 1; i++) {
    for(j = 1; j < n - 1; j++) {
       b[i * n + j] =
   (a[(i - 1) * n + j] +
    a[(i + 1) * n + j] +
    a[i * n + j - 1] +
    a[i * n + j + 1]) / 4.0;
    }
  }

  // Calcular la diferencia con respecto a la iteracion anterior
  double s = 0.0;
  for(i = 1; i < n - 1; i ++) {
    for(j = 1; j < n - 1; j++) {
s += fabs(a[i * n + j] - b[i * n + j]);
```

[1] Los detalles del cálculo de esta fórmula se pueden consultar en [4, p.329-330].

```
    }
  }

  // Si ha convergido, parar
  if (s < UMBRAL) {
    break;
  }

}

```

5.4.1. OpenMP

esquemas/difusion_omp.c

La paralelización OpenMP es directa porque la escritura en el bucle para calcular `b` en función de `a` no implica ninguna dependencia de datos.

Se puede obtener una versión algo más eficiente si se duplica la iteración para evitar la copia de `b` a `a` en cada iteración.

```
void iterar(double *a, double *b, int n) {
  #pragma omp parallel for private (i, j)
  for(int i = 1; i < n - 1; i++) {
    for(int j = 1; j < n - 1; j++) {
      b[i * n + j] =
        (a[(i - 1) * n + j] +
         a[(i + 1) * n + j] +
         a[i * n + j - 1] +
         a[i * n + j + 1]) / 4.0;
    }
  }
}

void calcular(double *a, double *b, int n) {
  // Iterar hasta el maximo de iteraciones o hasta que converja
  for(int it = 0; it < num_iteraciones; it++) {
    // Actualizar copiando el resultado en b
    iterar(a, b, n);
    // Actualizar de nuevo copiando el resultado en c
    iterar(b, a, n);

    // Calcular la diferencia con respecto a la iteracion anterior
    #pragma omp parallel for private (i, j) reduction(+:s)
    for(i = 1; i < n - 1; i ++) {
      for(j = 1; j < n - 1; j++) {
        s += fabs(a[i * n + j] - b[i * n + j]);
      }
    }
    ...
  }

```

5.4.2. MPI

esquemas/difusion_mpi.c

Implementación en MPI.

Existen varias descomposiciones de los datos. Una forma sencilla es realizar una descomposición por filas, de manera que cada proceso se encarga de n/p filas, donde n es el tamaño de la matriz y p el número de procesos.

El principal problema que hay que abordar es que para calcular los datos en los "bordes" es necesario acceder a los valores de otros procesos. Para minimizar el número de comunicaciones se pueden introducir "filas fantasma". La figura 5 muestra el esquema.

5.5. Ejemplo: Juego de la vida

El Juego de la vida es un autómata celular diseñado por el matemático británico John Horton Conway en 1970.

La configuración del juego es la siguiente:

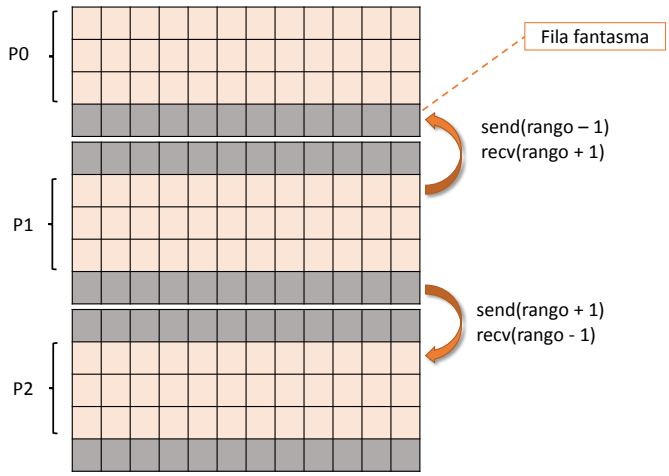


Figura 5: Organización de los datos en la implementación en MPI. Cada proceso incluye filas fantasma para albergar fácilmente los datos intercambiados con los vecinos.

- El tablero del juego es una matriz en el que cada celda tiene una célula.
- Cada célula tiene 8 células vecinas (se incluyen las diagonales).
- Las célula puede estar *viva* o *muerta*.
- Una célula está viva si tiene 3 vecinos vivos.
- Una célula está muerta en caso contrario.

El juego se "ejecuta" realizando iteraciones, y en cada iteración se establecen los valores de las células (1 para indicar que está viva o 0 para indicar que está muerta) según estas reglas:

- Una célula nace si tiene exactamente 3 células vecinas vivas.
- Una célula muerte si no tiene 2 o 3 células vecinas vivas (muere por soledad o por superpoblación).

```
while(true) {
  for (int fila = 0; fila <= n; fila++) {
    for (int col = 0; col <= m; col++) {
      int numVecinos = calculaVecinos(original, fila, col);
      if (original[fila][col]) {
        // Originalmente viva: ver si debe morir
        destino[fila][col] = true;
        if (numVecinos < 2)
          destino[fila][col] = false;
        if (numVecinos > 3)
          destino[fila][col] = false;
      } else {
        // Originalmente muerta: ver si debe nacer
        destino[fila][col] = false;
        if (numVecinos == 3)
          destino[fila][col] = true;
        }
    }
  }
}
```

6. Dependencias en árbol y grafos

6.1. Problema

Este patrón está relacionado con estructuras de datos en forma de árbol o grafo sobre las que se quiere aplicar un algoritmo en la que hay que manejar dependencias entre los nodos del árbol o el grafo. En este caso la paralelización no es tan directa porque no se puede dividir el trabajo fácilmente.

6.2. Solución

La principal aproximación para abordar este problema es reestructurar las operaciones del algoritmo para que conseguir exponer mayor concurrencia.

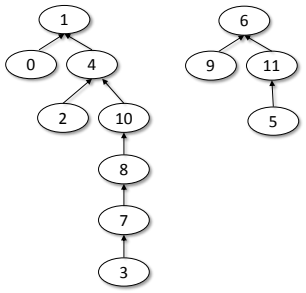
6.3. Ejemplo: Cálculo de la raíz de un bosque

Dado una estructura de datos que contiene n árboles, se desea encontrar el elemento raíz de cada nodo. Por ejemplo, la figura 6 muestra un ejemplo de un bosque con dos árboles. Este tipo de estructuras de datos pueden utilizarse, por ejemplo, para expresar relaciones de equivalencia entre elementos. Así, encontrar el elemento raíz equivale a encontrar el representante de la clase de equivalencia.

La implementación secuencial del algoritmo consiste en calcular iterativamente el *padre del padre* de un nodo. Ese nuevo padre se almacena como resultado parcial hasta que no haya más cambios. Si el padre del padre es igual padre actual, significa que ya se ha alcanzado la raíz del árbol.

El siguiente código en C muestra un extracto.

```c
int cambio = 1;
while (cambio) {
  cambio = 0;
  for(int i = 0; i < n; i++) {
    // Si el padre del nodo padre es distinto,
    // entonces "seguir" esa dependencia
    int padre_padre = nodos[nodos[i]];
    if (nodos[i] != padre_padre) {
      cambio = 1;
      nodos[i] = padre_padre;
    }
  }
}
```



Nodo	0	1	2	3	4	5	6	7	8	9	10	11
Padre	1	1	4	7	1	11	6	8	10	6	4	6

Figura 6: Ejemplo de bosque con dos árboles. Representación del árbol en forma de matriz. Basado en [1][p.350]

6.3.1. OpenMP

La implementación en memoria compartida consiste en paralelizar el bucle `for` para dividir el trabajo de establecer los padres.

El principal problema es garantizar que la actualización de `nodos[i]` se realiza de manera atómica (es decir, otro hilo no ha modificado al mismo tiempo el *padre del padre*).

Una alternativa sería crear un array de tantos `locks` como nodos del árbol hubiera. Se establecería una sección crítica que garantizara la lectura del padre del padre y su escritura de manera atómica. Sin embargo, en este caso la estrategia no es buena porque no se podría explotar casi ningún paralelismo.

Una alternativa es tener dos arrays de nodos. Uno de escritura y otro de lectura. Al final de cada iteración se intercambiarían. De esa manera, no es necesaria ninguna sincronización explícita (más allá de la barrera implícita al finalizar el `for` paralelo).

```c
int cambio = 1;
int iteracion = 0;
// nodos_r representa lo que se puede leer, nodos_w donde se escribe
int *nodos_r = nodos;
```

```
int *nodos_w = ... inicializar
memcpy(nodos_w, nodos, n);
while (cambio) {
  int cambio = 0;

  #pragma omp parallel for
  for(int i = 0; i < n; i++) {
    // Si el padre del nodo padre es distinto,
    // entonces "seguir" esa dependencia
    int padre = nodos_r[i];
    int padre_padre = nodos_r[padre];
    if (padre != padre_padre) {
      cambio = 1;
      nodos_w[i] = padre_padre;
    }
  }

  // intercambiar
  int *nodos_tmp = nodos_w;
  nodos_w = nodos_r;
  nodos_r = nodos_tmp;
}

// El resultado esta en nodos_r
nodos = nodos_r;
free(nodos_w);
```

7. Pipeline

7.1. Problema

La salida de una tarea es la entrada de la siguiente. Las tareas se pueden ejecutar en paralelo, pero el rendimiento global del pipeline está determinado por la tarea que tenga una mayor latencia (el tiempo que tarda en realizar su trabajo y entregarlo a la siguiente tarea).

7.2. Contexto

En una línea de ensamblaje (fig. 7) el proceso de construir un producto se divide en etapas. Cada etapa del proceso construye una componente del producto o se encarga de realizar alguna tarea sobre el mismo. Cada etapa tiene asociado un trabajador (puede ser un humano o un robot) que se encarga de llevarla a cabo. Tan pronto como el trabajador termina de realizar su trabajo, pasa el resultado a la siguiente etapa de la línea, y toma los materiales que necesita para volver a realizar su tarea.

La ventaja de una línea de ensamblaje *pipeline* es que una vez que todas las etapas tienen trabajo, permanecen ocupadas realizando trabajo útil. La princpal restricción es que que no haya ninguna etapa que tarde mucho más que las demás, ya que la etapa más lenta determina el rendimiento global del pipeline.

Esta analogía resulta útil en el contexto de la programación paralela puesto que ciertos problemas se pueden organizar como una serie de etapas sucesivas en las que una etapa depende de la anterior.

Por ejemplo, dado un conjunto de imágenes a las que se les quiere aplicar operaciones de transformación de manera sucesiva, se podría crear un pipeline en el que la primera etapa carga la imagen, la segunda etapa aplica un filtro, la tercera etapa la escala, etc.

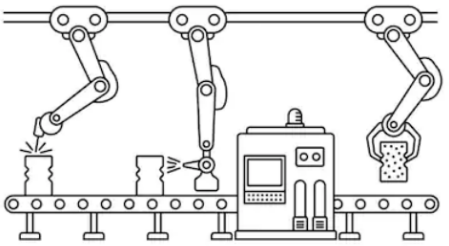


Figura 7: Ejemplo de línea de ensamblaje. Tomada de `http://shutterstock.com`.

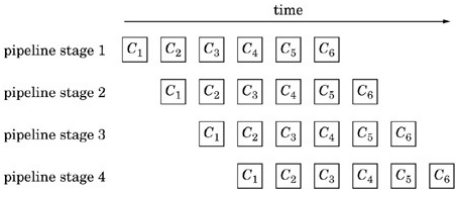


Figura 8: Organización de un *pipeline*. Cada etapa se encarga de una parte del cálculo. Cuando una etapa terminar de realizar el trabajo C_i, le pasa el resultado actual a la siguiente etapa. Imagen tomada de [3][p.150]

7.3. Solución

En primer lugar se deben identificar las etapas o tareas que va a realizar el *pipeline*. Estas etapas son fijas y no cambian durante la ejecución. Cada etapa

tendrá asignado un trabajador (hilo o proceso) que se encargará de realizar su trabajo. Recibirá datos de la etapa anterior y enviará su resultado a la etapa siguiente. La figura 8 muestra un ejemplo.

7.3.1. OpenMP y Java

En el modelo de programación de memoria compartida la gestión del pipeline debe realizarse explícitamente implementando algún sistema de colas para que cada tarea del pipeline reciba los resultados y los envíe al siguiente.

7.3.2. MPI

En un sistema de paso de mensajes la implementación de este patrón es sencilla. Cada tarea del pipeline se asigna a un proceso. El primer proceso se encarga de procesar o generar los datos y los va enviando uno a uno al siguiente proceso. Este proceso realiza su tarea y envía el resultado al siguiente proceso, etc.

La principal cuestión es cómo enviar una marca de fin. Para ello, dependiendo del tipo de mensaje se puede utilizar valor especial (ej., -1 si son números positivos) o construir un mensaje compuesto (ej., una estructura) en el que exista un *flag* para indicar la marca de fin.

A modo de ejemplo muy simple, supongamos un pipeline en el que se desea calcular qué números son impares (una etapa) y qué números son son múltiplos de 7. El esquema del pipeline sería similar al siguiente:

```
if (rank == 0) {
  for(int i=0; i < N; i++) {
    MPI_Send(&i, 1, MPI_INT, 1, 0, MPI_COMM_WORLD);
  }
  int marca = -1;
  MPI_Send(&marca, 1, MPI_INT, 1, 0, MPI_COMM_WORLD);
} else if (rank == 1) {
  int num;
  while (true) {
    MPI_Recv(&num, 1, MPI_INT, 0, 0, MPI_COMM_WORLD);
    if (num == -1)
      break;
    if (num % 2 != 0) {
      MPI_Send(&num, 1, MPI_INT, 2, 0, MPI_COMM_WORLD);
    }
  }
} else if (rank == 2) {
  int num;
  while (true) {
    MPI_Recv(&num, 1, MPI_INT, 1, 0, MPI_COMM_WORLD);
    if (num == -1)
      break;
    if (num % 7 == 0) {
      printf("%d cumple los requisitos\n", num);
    }
  }
}
```

Unix pipes

El operador *pipe* de Unix es la forma en que se construyen pipelines de comandos. Por ejemplo,

```
cat fichero.txt | grep Juan | wc --lines
```

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.

[3] T. G. Mattson, B. Sanders, and B. Massingill. *Patterns for parallel programming*. Pearson Education, 2004.

[4] M. J. Quinn. Parallel programming in c with mpi and openmp,(2003).