Paral·lelisme

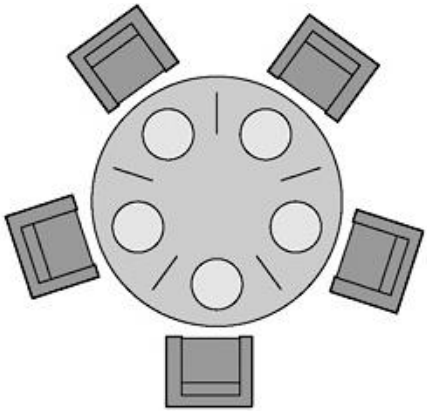
FIB-UPC

Collection of exercises

© http://fibernalia.blogspot.com

(1 & 2) Concurrency and parallelism

1.



Si dos filòsofs adjacents intenten agafar alhora el bastonet que hi ha entre tots dos, un d'ells es quedarà sense menjar (**condició de carrera**).

Si els cinc filòsofs agafen alhora el bastonet que hi ha a la seva esquerra, cadascun d'ells es quedarà esperant a que el filòsof que està assegut a la seva dreta alliberi l'altre bastonet que li cal per menjar (***deadlock***).

Possibles solucions

● Limitar a quatre el nombre de filòsofs que puguin estar asseguts a taula, on únicament menjaran.

● Menjar per torns: primer un, després el filòsof de la dreta i així successivament, seguint un ordre cíclic.

● Si un filòsof A té gana, si té els dos bastonets, menja; si no en té cap, s'espera a que els seus respectius filòsofs adjacents B i C els alliberin; si en té un, per exemple el que està entre A i B, i després d'un temps **aleatori** C no allibera l'altre, A allibera el seu i, si B no estava a la cua per agafar-lo, el torna a agafar i torna a esperar-se un temps aleatori.

© http://fibernalia.blogspot.com

Would it be a solution if philosophers put down a fork after waiting five minutes and wait a further five minutes before making their next attempt?

No, perquè si tots agafen alhora el bastonet que hi ha a la seva esquerra, esperaran cinc minuts, alliberaran el bastonet i al cap de cinc minuts el tornaran a agafar. Estaríem davant d'un problema de ***livelock*** (*two or more tasks continuously change their state in response to changes in the other tasks without doing any useful work*).

Solució: temps d'espera aleatori.

```
import java.io.*;
import java.util.*;

public class Philosopher extends Thread {
    static final int count = 5;
    Chopstick left;
    Chopstick right;
    int position;

    Philosopher(int position,
                Chopstick left,
                Chopstick right) {
        this.position = position;
        this.left = left;
        this.right = right;
    }
```

```
public static void main(String[] args) {
        Philosopher phil[] = new Philosopher[count];

        Chopstick last = new Chopstick();
        Chopstick left = last;
        for(int i=0; i<count; i++){
            Chopstick right = (i==count-1)?last :
                                    new Chopstick();
            phil[i] = new Philosopher(i, left, right);
            left = right;
        }

        for(int i=0; i<count; i++){
            phil[i].start();
        }
    }

    ...
}
```

```
public void run() {
        try {
            while(true) {
1                   synchronized(left) {
2                       synchronized(right) {
3                           System.out.println(times + ": Philosopher " + position + " is done eating");
                        }
                    }
            }
        } catch (Exception e) {
            System.out.println("Philosopher " + position + "'s meal got disturbed");
        }
    }
```

© http://fibernalia.blogspot.com

```
static Object table;
public void run() {
      try {
         while(true) {
1                synchronized(table) {
2                   synchronized(left) {
3                      synchronized(right) {
4                         System.out.println(times + ": Philosopher " + position + " is done eating");
                       }
                   }
               }
         }
      } catch (Exception e) {
         System.out.println("Philosopher " + position + "'s meal got disturbed");
      }
   }
```

```
public void run() {
    try {
        Chopstick first = (position%2 == 0)?left:right;
        Chopstick second = (position%2 == 0)?right:left;
            while(true) {
1                synchronized(first) {
2                   synchronized(second) {
3                      System.out.println(times + ": Philosopher " + position + " is done eating
                    }
                }
            }
        } catch (Exception e) {
            System.out.println("Philosopher " + position + "'s meal got disturbed");
        }
    }
```

Task 1

```
for (int i = 0; i < M; ++i)
        for (int j = 0; j < N; ++j)
            A[i][j] = B[i][j] = i;
```

Task 2

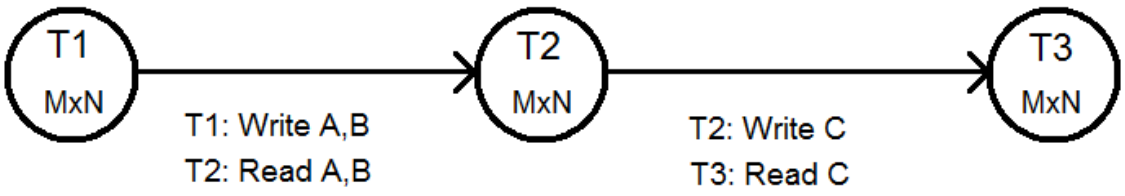
```
for (int i = 0; i < M; ++i)
        for (int j = 0; j < N; ++j)
            C[i][j] = A[i][j] + B[i][j];
```

Task 3

```
for (int i = 0; i < M; ++i)
        for (int j = 0; j < N; ++j)
            printf ("%d \n", C[i][j]);
```

* Cost de cada iteració: 1 unitat de temps

© http://fibernalia.blogspot.com



$T_1 = 3*M*N$ $T_\infty = 3*M*N$ $P = T_1 / T_\infty = 1$ $P_{min} = 1$

Task 1

```
for (int i = 0; i < M; ++i)
    for (int j = 0; j < N; ++j)
        A[i][j] = B[i][j] = i;
```

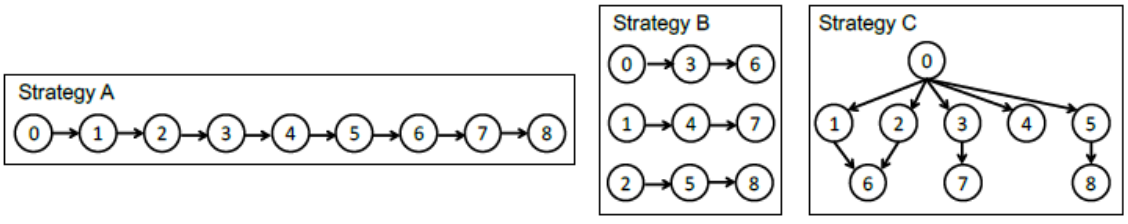
Task 2

```
for (int i = 0; i < M; ++i)
    for (int j = 0; j < N; ++j)
        C[i][j] = A[i][j] + B[i][j];
```

Task 3

```
for (int i = 0; i < M; ++i)
    for (int j = 0; j < N; ++j)
        printf ("%d \n", C[i][j]);
```

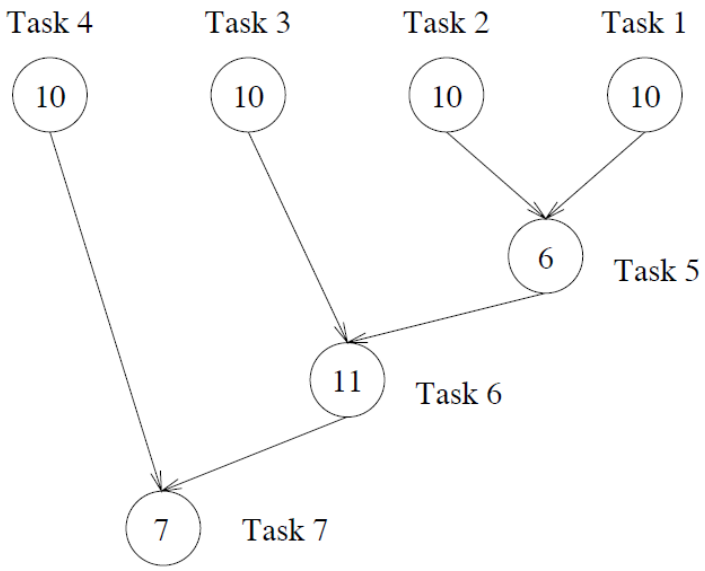
$T_1 = 3*M*N$ $T_\infty = N + 1 + M*N$ $P = T_1 / T_\infty$ $P_{min} = M*N$



* Cost de cada tasca: 1 unitat de temps

Strategy A: $T_1 = 9, T_\infty = 9,$ `Parallelism`$= 1,\ P_{min} = 1.$
Strategy B: $T_1 = 9, T_\infty = 3,$ `Parallelism`$= 3,\ P_{min} = 3.$
Strategy C: $T_1 = 9, T_\infty = 3,$ `Parallelism`$= 3,\ P_{min} = 5.$

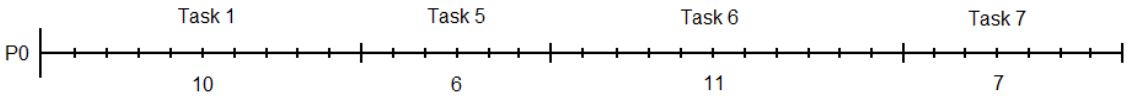
© http://fibernalia.blogspot.com



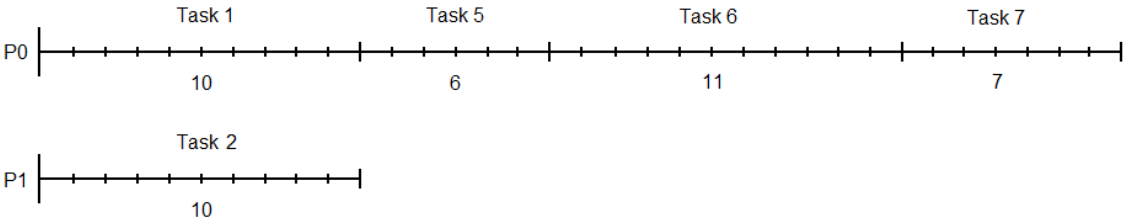
$T_1 = 10*4 + 6 + 11 + 7 = 64$ $T_\infty = 10 + 6 + 11 + 7 = 34$ (camí crític)

$P = 64 / 34 = 1.88$ P_{min} ?

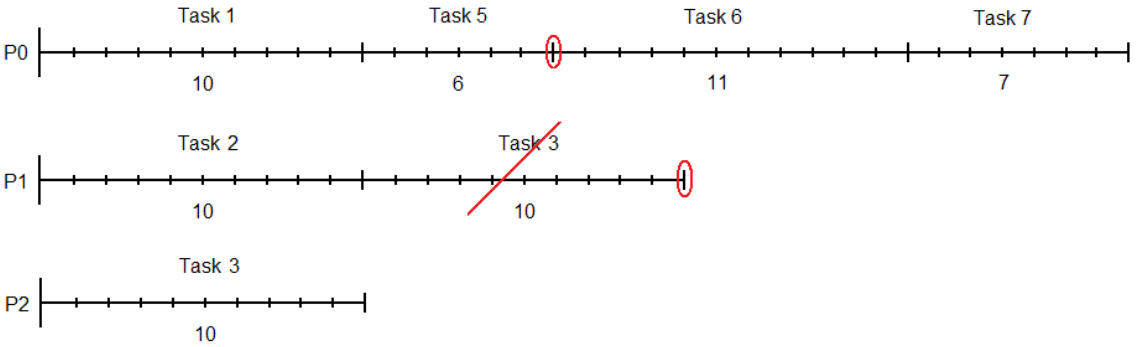
1) Camí crític a P0



2) Task 5 depèn de Task 2

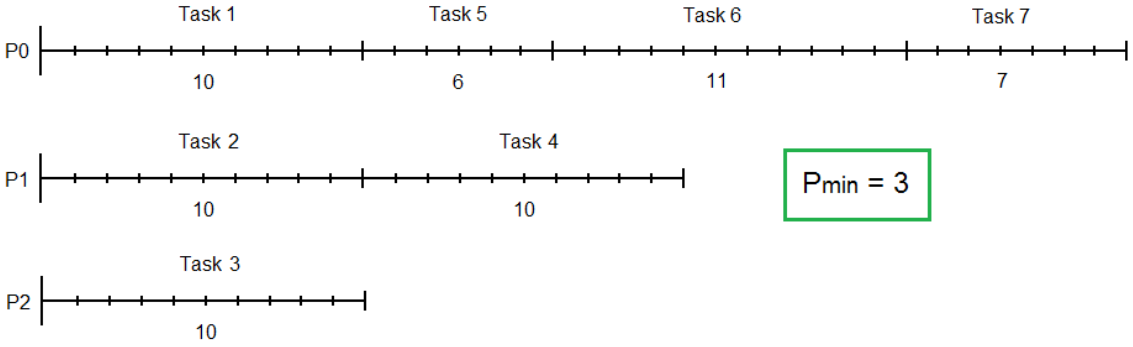


3) Task 6 depèn de Task 3

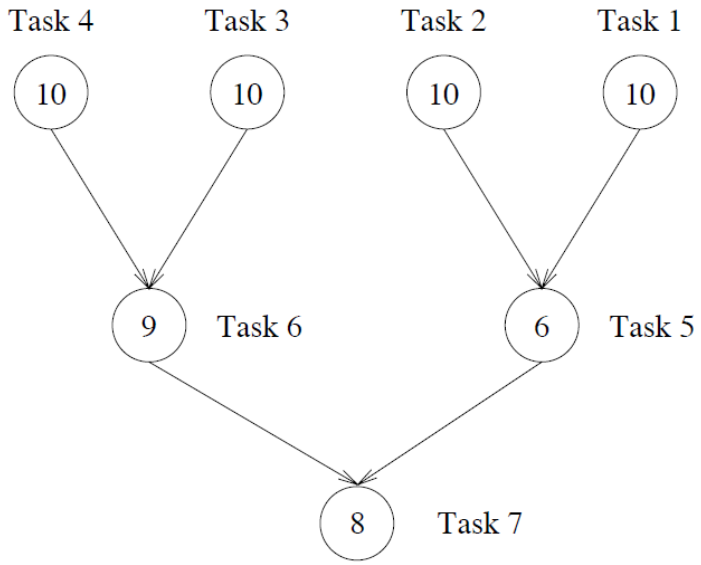


© http://fibernalia.blogspot.com

4) Task 7 depèn de Task 4 (pot anar tant a P1 com a P2)

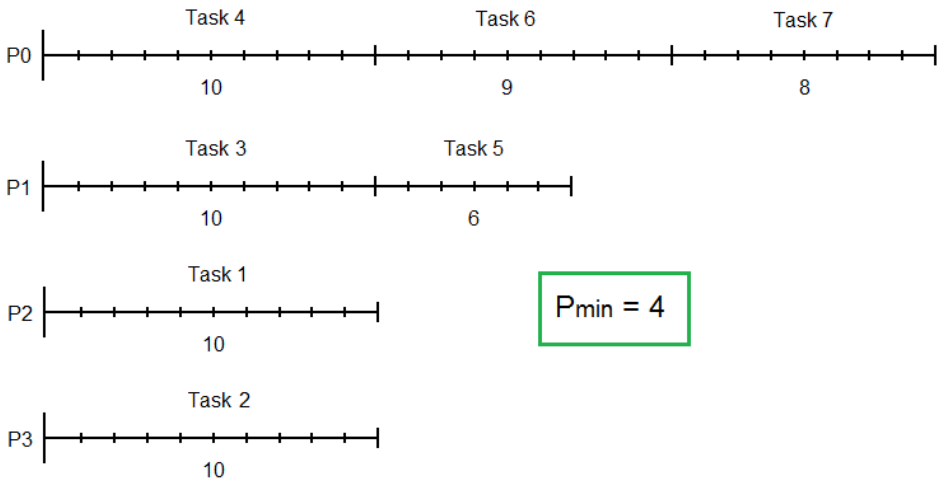


Pmin = 3



$T_1 = 10*4 + 9 + 6 + 8 = 63$ $T_\infty = 10 + 9 + 8 = 27$ (camí crític) $P = 63 / 27 = 2.33$

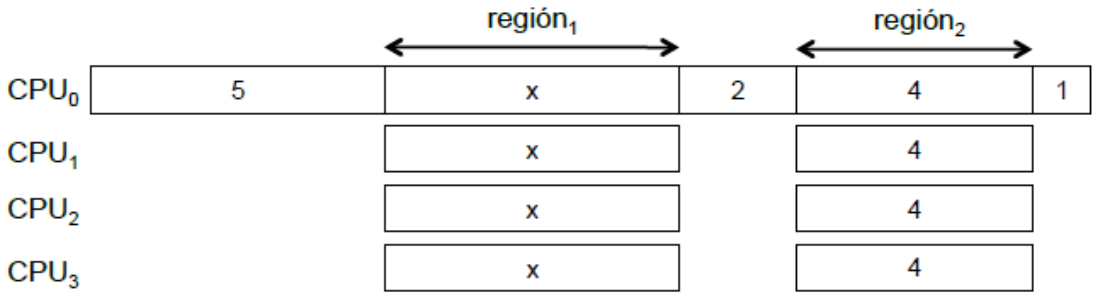
P_{min} ?



Pmin = 4

© http://fibernalia.blogspot.com

2.

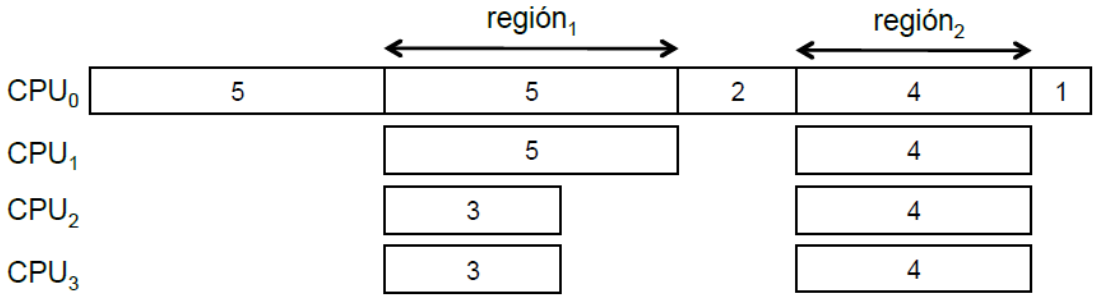


(a) $S_\infty = 1 / (1 - \varphi) \rightarrow 9 = 1 / (1 - \varphi) \rightarrow \varphi = 8/9$

(b) $S_p = 1 / ((1 - \varphi) + (\varphi / p)) \rightarrow S_4 = 1 / ((1 - 8/9) + (8/9 / 4)) = 3$

(c) $S_p = T_1 / T_p \rightarrow S_4 = T_1 / T_4 = (5 + 4X + 2 + 4*4 + 1) / (5 + X + 2 + 4 + 1) =$
$= 3 \rightarrow X = 12$

3.



(a) $S_4 = T_1 / T_4 = 40 / 17 = 2.35$ $\varphi = 32 / 40 = 0.8$ $S_\infty = 1 / (1 - 0.8) = 5$

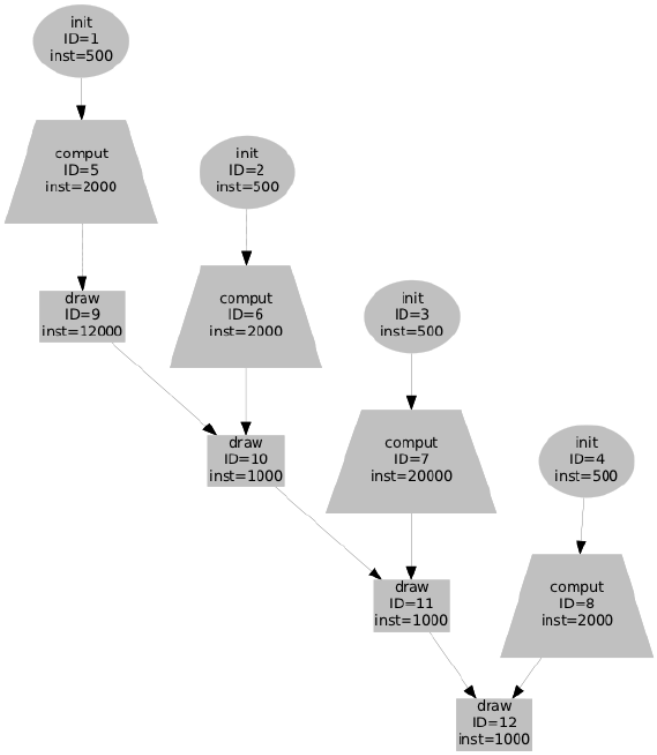
5.

(a) $T_1 = 500*4 + 2000*3 + 20.000 + 12.000 + 1000*3 = 43.000$

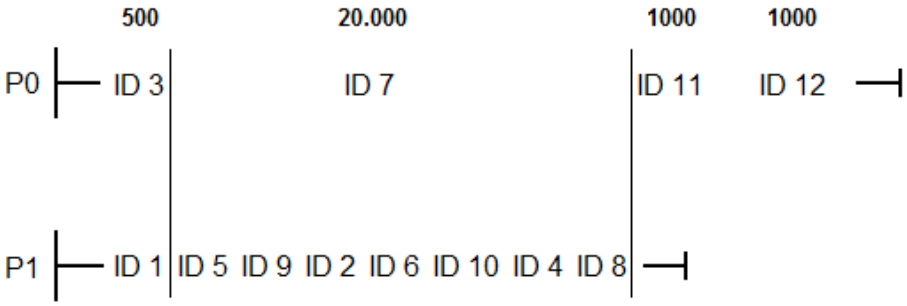
$T_\infty = 500 + 20.000 + 1000 + 1000 = 22.500$

$P = T_1 / T_\infty = 1.91$

© http://fibernalia.blogspot.com



(b) $P_{min} = 2$



(c) $T_1 = 500*4 + 6500*4 + 12.000 + 1000*3 = 43.000$

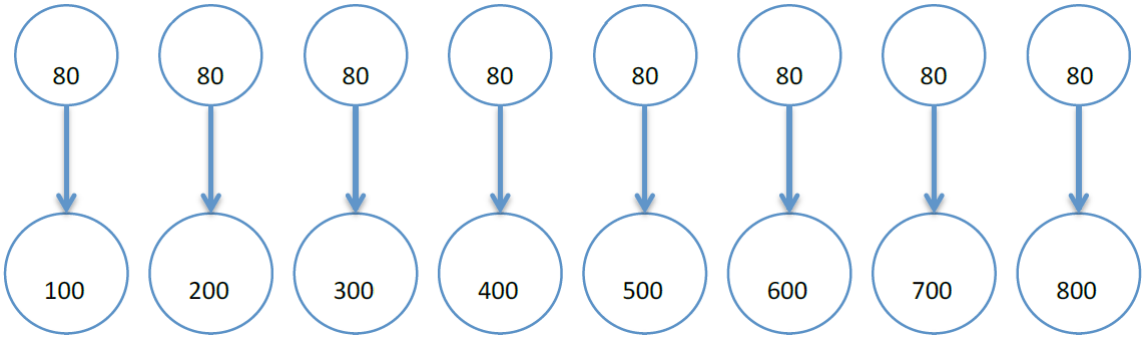
$T_\infty = 500 + 6500 + 12.000 + 1000*3 = 22.000$

$P = T_1 / T_\infty = 1.95$

© http://fibernalia.blogspot.com

6.

(a)



(b) $T_1 = 80*8 + 100 + 200 + ... + 800 = 4240$

$T_\infty = 80 + 800 = 880$

$P = T_1 / T_\infty = 4.82$ $P_{min} = 5$

(c) P0: 80,80,100,800 P1: 80,80,200,700 P2: 80,80,300,600

P3: 80,80,400,500

$S_4 = T_1 / T_4 = 4240 / 1060 = 4$

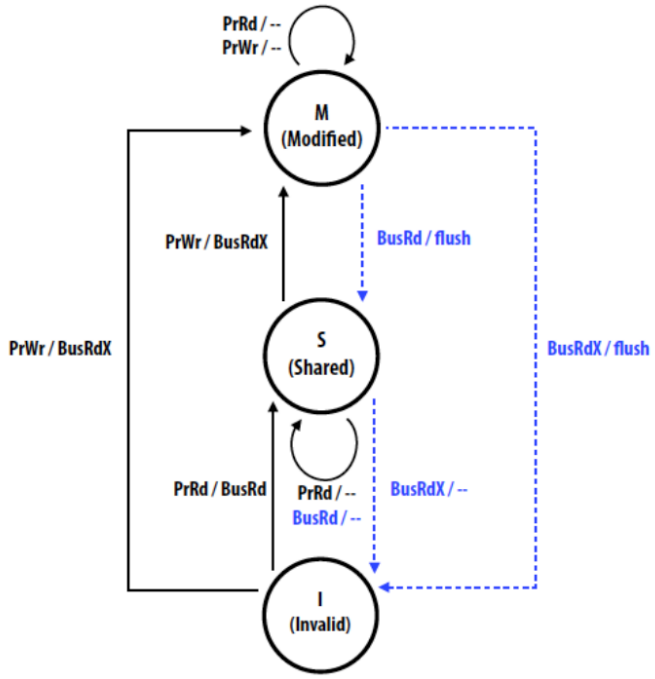
© http://fibernalia.blogspot.com

11.

		Column distribution	Row distribution
Initial communication	Total number of messages	$P-1$	-
	Size of each message	N	-
	Contribution to T_p	$t_s + N \times t_w$	-
Blocks calculation (B)	Number of blocks	1 per thread	N/B per thread
	Number of elements in a block	$N \times N/P$	$B \times N/P$
	Contribution to T_p	$N \times N/P \times t_c$	$(N/B + P - 1) \times (B \times N/P) \times t_c$
Communication during the parallel calculus	Total number of messages	-	$N/B * (P-1)$
	Size of each message	-	B
	Contribution to T_p	-	$(N/B+P-2) \times (t_s + B \times t_w)$

© http://fibernalia.blogspot.com

3. Multiprocessor architectures



1.

Memory Access	CPU event	Bus transaction(s)	Cache Line State		
			Mem1	Mem2	Mem3
r1	PrRd	BusRd	S (E)	I	I
w1	PrWr	BusRdX (-)	M	I	I
r2	PrRd	BusRd + Flush	S	S	I
w3	PrWr	BusRdX	I	I	M
r2	PrRd	BusRd + Flush	I	S	S
w1	PrWr	BusRdX	M	I	I
w2	PrWr	BusRdX + Flush	I	M	I
r3	PrRd	BusRd + Flush	I	S	S
r2	PrRd	-	I	S	S
r1	PrRd	BusRd	S	S	S

■ MESI

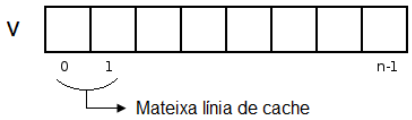
© http://fibernalia.blogspot.com

2.

(a)

Memory Access	CPU event	Bus transaction(s)	Cache Line State		
			Mem1	Mem2	Mem3
w1	PrWr	BusRdX	M	I	I
r2	PrRd	BusRd + Flush	S	S	I
w3	PrWr	BusRdX	I	I	M

(b)



Memory Access	CPU event	Bus transaction(s)	Cache Line State		
			Mem1	Mem2	Mem3
r1 (v[0])	PrRd	BusRd	S	I	I
r2 (v[1])	PrRd	BusRd	S	S	I
w1 (v[0])	PrWr	BusRdX	M	I	I

FALSE SHARING

4. Overhead de crear els threads, de sincronitzar-los al final del for i de false sharing.

5.

(a) True

(b) False

(c) True

(d) False

(e) True

(f) True

(g) False

(h) False

© http://fibernalia.blogspot.com

7.

(a) ii. ... tiene acceso a 32 GB de memoria física

(b) iv. Ninguna de las anteriores es cierta

(c) ii. ... lo realiza en sistema operativo en base a una política predefinida

(d) i. ... coincide con el número de líneas que caben en su memoria principal asociada

(e) i. ... las instrucciones atómicas bloquean al procesador que las invoca en caso de que la posición de memoria a la que se quiere acceder este siendo accedida por otro procesador

(f) iii. ... se agrava cuando ocurre entre NUMAnodes, debido a la no uniformidad en el tiempo de acceso a memoria

8.

(a) False

(b) False

(c) False

(d) False

(e) False

(f) True

(g) False

(h) False

(i) False

(j) False

© http://fibernalia.blogspot.com

4. Task decomposition

1. `#pragma omp for nowait`

2. Problema: False sharing. Solució: Padding

8.

(a) Versió 1

```
int sum_vector(int *X, int n) {
  int i;
  int sum = 0;
  #pragma omp parallel for reduction (+ : sum)
  for (i = 0; i < n; ++i) sum += X[i];
  return sum;
}
```

Versió 2

```
int lsum[NTHREADS][BYTES_CACHE_LINE / sizeof(int)];
int sum_vector(int *X, int n) {
  int i;
  int sum = 0;
  #pragma omp parallel for
  for (i = 0; i < n; ++i)
    lsum[omp_get_thread_num()][0] += X[i];
  for (i = 0; i < omp_get_num_threads(); ++i)
    sum += lsum[i][0];
```

© http://fibernalia.blogspot.com

```
  return sum;
}
```

(b) i (c)

```
int recursive_sum_vector(int *X, int n, int d) {
  int tmp1, tmp2;
  if (n>MIN_SIZE) {
    int n2 = n / 2;
    #pragma omp task shared(tmp1) final(d>MAXDEPTH) mergeable
    tmp1 = recursive_sum_vector(X, n2, d+1);
    #pragma omp task shared(tmp2) final(d>MAXDEPTH) mergeable
    tmp2 = recursive_sum_vector(X+n2, n-n2, d+1);
    #pragma omp taskwait
    return(tmp1+tmp2);
  }
  else return(sum_vector(X, n));
}
void main() {
  ...
  #pragma omp parallel
  #pragma omp single
  sum = recursive_sum_vector(v,N,0);
  ...
}
```

© http://fibernalia.blogspot.com

9.

(a)

```
void FindBounds(int * input, int size, int * min, int * max) {
  int tmin=*min, tmax=*max;
  #pragma omp parallel reduction(max: tmax) reduction(min: tmin)
  for (int i=0; i<size; ++i){
    if (input[i]>(tmax)) (tmax)=input[i];
    if (input[i]<(tmin)) (tmin)=input[i];
  }
  *min=tmin; *max=tmax; }
```

(b)

```
#pragma omp parallel for
for (int i=0; i<size; ++i) {
    tmp = ((input[i] - min) * HIST_SIZE / (max - min - 1));
    #pragma omp atomic
    ++histogram[tmp];
}
```

(c)

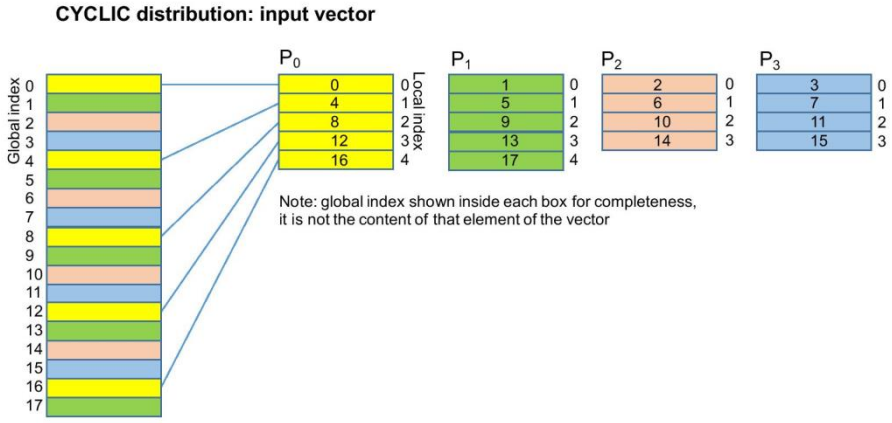
```
omp_lock_t histlock[HIST_SIZE];
for (int i=0; i<HIST_SIZE; ++i) omp_init_lock(&histlock[i]);
#pragma omp parallel for
for (int i=0; i<size; ++i) {
    tmp = ((input[i] - min) * HIST_SIZE / (max - min - 1));
    omp_set_lock (&histlock[tmp]);
    ++histogram[tmp];
    omp_unset_lock (&histlock[tmp]);
}
for (int i=0; i<HIST_SIZE; ++i) omp_destroy_lock(&histlock[i]);
```

© http://fibernalia.blogspot.com

5. Data decomposition

(Pseudo-)Problema 6 (ampliado para considerar memoria distribuida)

a1) Función `FindBounds`, data decomposition CYCLIC para vector `input`. Arquitectura memoria compartida.




```
void FindBounds(int * input, int size, int * min, int * max) {
int tmin=*min, tmax=*max; // reducción no permitida sobre punteros

#pragma omp parallel reduction(max: tmax) reduction(min: tmin)
      {
      int i_start = omp_get_thread_num();
      int howmany = omp_get_num_threads();
      for (int i=i_start; i<size; i += howmany) {
            if (input[i]>(tmax)) (tmax)=input[i];
            if (input[i]<(tmin)) (tmin)=input[i];
            }
      }
*min=tmin; *max=tmax;
}
```

a2) Función `FindBounds`, data decomposition CYCLIC para vector `input`. Arquitectura memoria distribuida. Versión OpenMP (no valido como código paralelo, sólo para entender como cambia la indexación de las estructuras de datos).

```
void FindBounds(int * input, int size, int * min, int * max) {
int tmin=*min, tmax=*max; // reducción no permitida sobre punteros

#pragma omp parallel reduction(max: tmax) reduction(min: tmin)
      {
      int who = omp_get_thread_num();
      int howmany = omp_get_num_threads();
      int rem = size % howmany;
      int i_end = (size / howmany) + (who < rem ? 1 : 0);
      for (int i=0; i<i_end; i++) {
            if (input[i]>(tmax)) (tmax)=input[i];
            if (input[i]<(tmin)) (tmin)=input[i];
            }
      }
*min=tmin; *max=tmax;
}
```

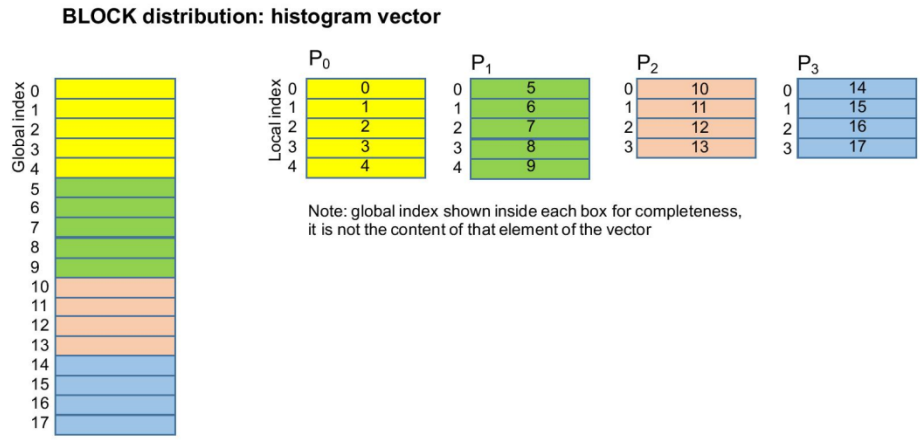
© http://fibernalia.blogspot.com

a3) Función `FindBounds`, data decomposition CYCLIC para vector `input`. Arquitectura memoria distribuida. Versión MPI (no incluye operación de reducción, ver apartado c)).

```
void FindBounds(int * input, int size, int * min, int * max) {
int who, howmany;

MPI_Comm_rank(MPI_COMM_WORLD, &who);
MPI_Comm_size(MPI_COMM_WORLD, &howmany);
int rem = size % howmany;
int i_end = (size / howmany) + (who < rem ? 1 : 0);
for (int i=0; i<i_end; i++) {
      if (input[i]>(*max)) (*max)=input[i];
      if (input[i]<(*min)) (*min)=input[i];
       }
}
```

b1) Función `FindFrequency`, data decomposition BLOCK para vector `histogram`. Arquitectura memoria compartida




```
void FindFrequency(int * input, int size ,
                   int * histogram, int min, int max) {
#pragma omp parallel private(tmp)
    {
    int who = omp_get_thread_num();
    int howmany = omp_get_num_threads();
    int i_start = who * (HIST_SIZE/howmany) +
              (who < (HIST_SIZE%howmany) ? who : (HIST_SIZE%howmany));
    int i_end = i_start + (HIST_SIZE/howmany) +
              (who<(HIST_SIZE%howmany));
    for (int i=0; i<size; i++) {
        int tmp = (input[i] - min) * HIST_SIZE / (max - min - 1);
        if ((tmp>=i_start) && (tmp<i_end))
            histogram[tmp]++;
        }
    }
```

© http://fibernalia.blogspot.com

b2) Función `FindFrequency`, data decomposition BLOCK para vector `histogram`. Vector input replicado. Arquitectura memoria distribuida. Versión OpenMP (no valido como código paralelo, sólo para entender como cambia la indexación de las estructuras de datos).

```
void FindFrequency(int * input, int size ,
                   int * histogram, int min, int max) {

#pragma omp parallel
    {
    int who = omp_get_thread_num();
    int howmany = omp_get_num_threads();
    int i_start = who * (HIST_SIZE/howmany) +
                (who < (HIST_SIZE%howmany) ? who : (HIST_SIZE%howmany));
    int i_end = i_start + (HIST_SIZE/howmany) +
                (who<(HIST_SIZE%howmany));
    for (int i=0; i<size; i++) {
        int tmp = (input[i] - min) * HIST_SIZE / (max - min - 1);
        if ((tmp>=i_start) && (tmp<i_end))
            histogram[tmp-i_start]++;
        }
    }
}
```

b3) Función `FindFrequency`, data decomposition BLOCK para vector `histogram`. Vector input replicado. Arquitectura memoria distribuida. Versión MPI (no incluye operación colectiva de gather, ver apartado c)).

```
void FindFrequency(int * input, int size ,
                   int * histogram, int min, int max) {
int who, howmany;

MPI_Comm_rank(MPI_COMM_WORLD, &who);
MPI_Comm_size(MPI_COMM_WORLD, &howmany);

int i_start = who * (HIST_SIZE/howmany) +
              (who < (HIST_SIZE%howmany) ? who : (HIST_SIZE%howmany));
int i_end   = i_start + (HIST_SIZE/howmany) +
              (who<(HIST_SIZE%howmany));
for (int i=0; i<size; i++) {
      int tmp = (input[i] - min) * HIST_SIZE / (max - min - 1);
      if ((tmp>=i_start) && (tmp<i_end))
            histogram[tmp-i_start]++;
      }
}
```

c) Para `frequency` sería necesario que P_0 distribuyera el vector entre todos los procesadores (comunicación colectiva tipo “scatter”). Dado que no se inicializa, también sería valido si no se realiza esta comunicación. Sin embargo, es necesario que P_0 recoja del resto de procesadores la porción del vector frequency calculado (colectiva tipo “gather”) después de la ejecución de `FindFrequency`.

Para la variable `max` es necesario que P_0 la replique en el resto de procesadores (comunicación colectiva tipo “broadcast”) antes de la ejecución de `FindBounds` (dado que está inicializada). Después de la ejecución de `FindBounds` P_0 deberá combinar los máximos parciales en cada procesador (comunicación colectiva tipo “reduce”) y volver a hacer un “broadcast” antes de iniciar la ejecución de `FindFrequency` con el objetivo de distribuir a todos los procesadores el valor máximo encontrando. Esta operación colectiva “reduce-broadcast” conjunta también existe en MPI y se denomina “allreduce”.

© http://fibernalia.blogspot.com