

ESTRUCTURA DE COMPUTADORES II

PRÁCTICAS DE MEMORIAS



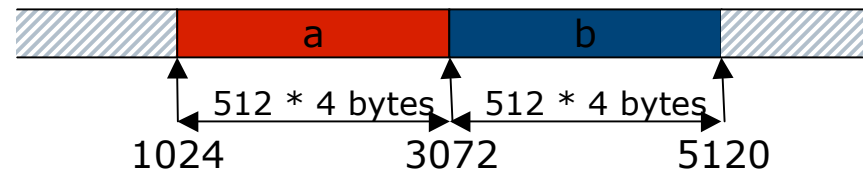
Técnicas de Optimización Software

ORIENTACIÓN

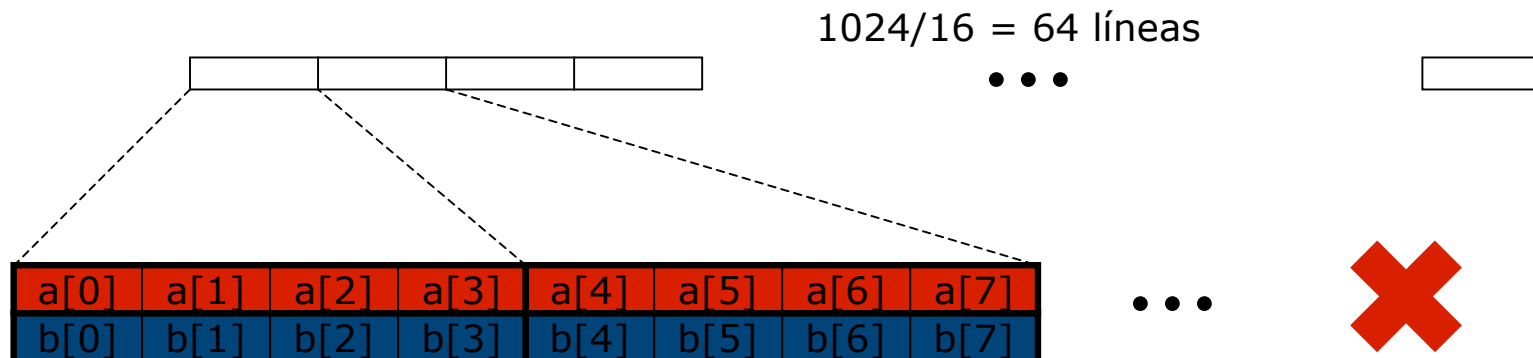
Fusión de arrays: Problema

```
float a[512], b[512];

for(i=0; i<512; i++)
    a[i] += b[i];
```

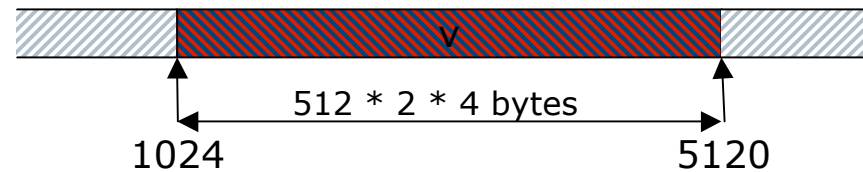


Caché de 1024 bytes, líneas de 16 bytes, correspondencia directa

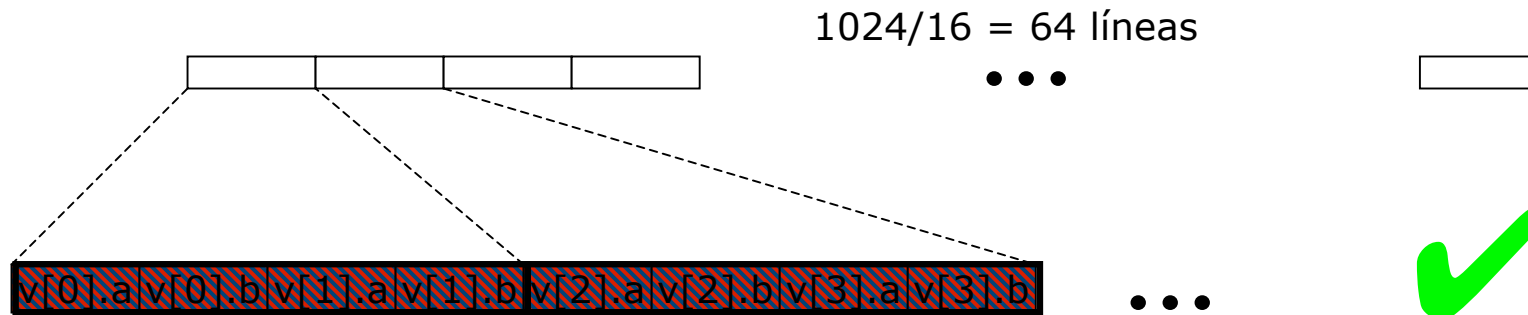


Fusión de arrays: Solución

```
struct reg {  
    float a, b;  
};  
struct reg v[512];  
  
for(i=0; i<512; i++)  
    v[i].a += v[i].b;
```

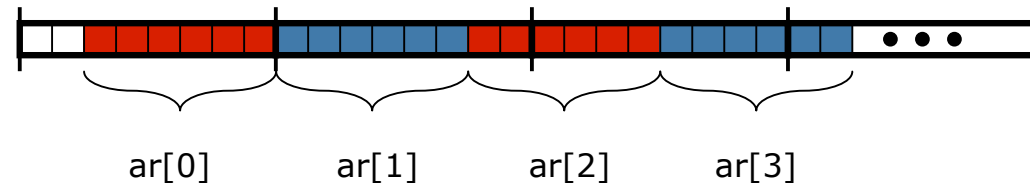


Caché de 1024 bytes, líneas de 16 bytes, correspondencia directa



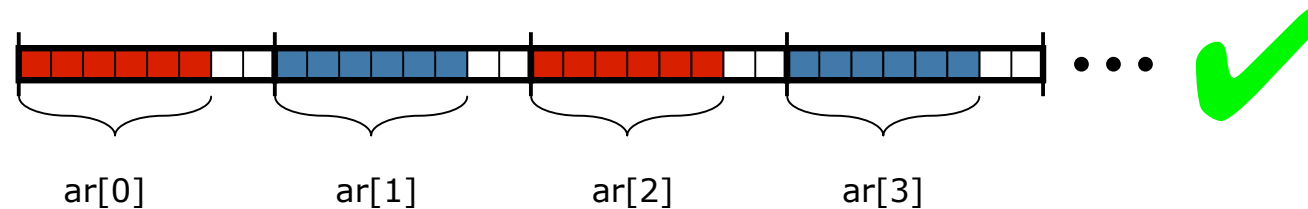
Rellenado y alineado de estructuras: Problema

```
struct ex {  
    float a;  
    short int b;  
};  
struct ex ar[512];  
  
for(i=0; i<512; i++) {  
    r += ar[d[i]].a;  
    t += ar[d[i]].b;  
}
```



Rellenado y alineado de estructuras: Solución

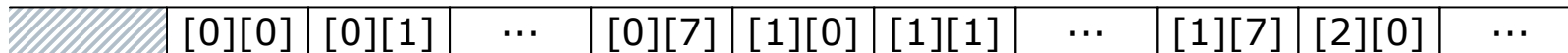
```
struct ex {  
    float a;  
    short int b;  
    char relleno[2];  
};  
...  
ar = (struct ex *) malloc(sizeof(struct ex) * 512 + BLOCK);  
ar = (struct ex *) ((( (int) ar + BLOCK -1) / BLOCK ) * BLOCK);  
...  
for(i=0; i<512; i++) {  
    r += ar[d[i]].a;  
    t += ar[d[i]].b;  
}
```



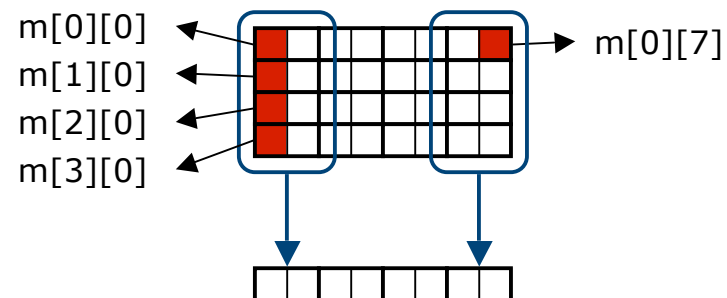
Rellenado de arrays: Problema

```
double m[4][8];  
  
for(j=0; j < 8; j++)  
    for(i=0; i < 4; i++)  
        q = q * 1.2 + m[i][j];
```

C almacena por filas;
FORTRAN por columnas



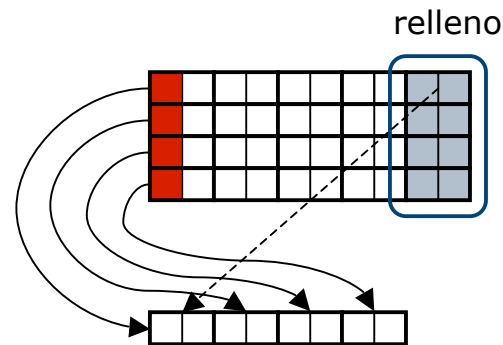
Caché de 64 bytes, líneas de 16 bytes, correspondencia directa



Rellenado de arrays: Solución

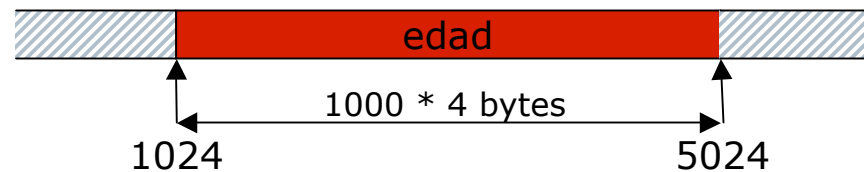
```
double m[4][10];  
  
for(j=0; j < 8; j++)  
    for(i=0; i < 4; i++)  
        q = q * 1.2 + m[i][j];
```

Caché de 64 bytes, líneas de 16 bytes, correspondencia directa



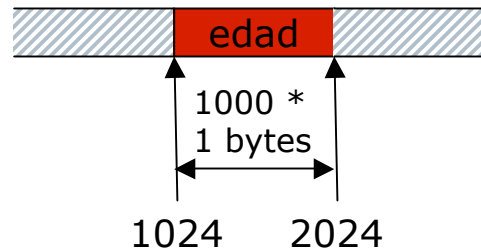
Empaquetado: Problema

```
int edad[1000], media=0;  
  
for(j=0; j < 1000; j++)  
    media += edad[j];  
media /= 1000;
```



Empaquetado: Solución

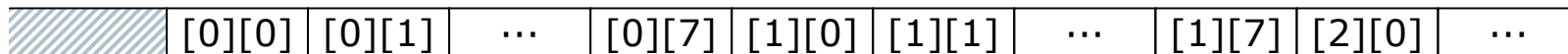
```
char edad[1000];  
int media=0;  
  
for(j=0; j < 1000; j++)  
    media += edad[j];  
media /= 1000;
```



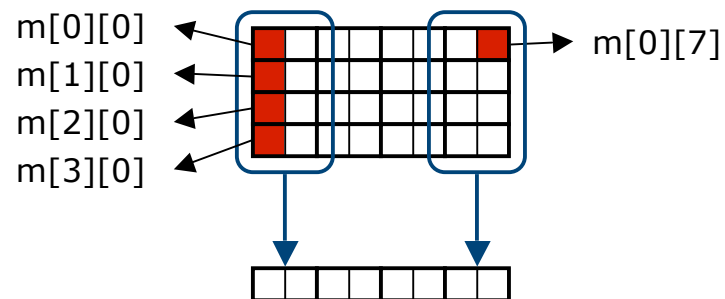
Intercambio de bucles: Problema

```
double m[4][8];  
  
for(j=0; j < 8; j++)  
    for(i=0; i < 4; i++)  
        q += m[i][j];
```

C almacena por filas;
FORTRAN por columnas

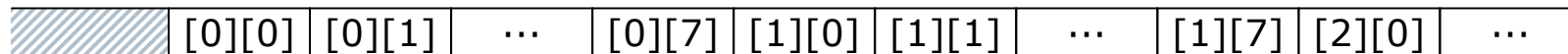


Caché de 64 bytes, líneas de 16 bytes, correspondencia directa

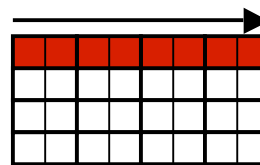


Intercambio de bucles: Solución

```
double m[4][8];  
  
for(i=0; i < 4; i++)  
    for(j=0; j < 8; j++)  
        q += m[i][j];
```



Caché de 64 bytes, líneas de 16 bytes, correspondencia directa



Fusión de bucles: Problema

```
for( i=0; i<N; i++)  
  for( j=0; j<N; j++)  
    a[i][j] = 1 / b[i][j] * c[i][j];  
  
for( i=0; i<N; i++)  
  for( j=0; j<N; j++)  
    d[i][j] = a[i][j] + c[i][j];
```



La caché al finalizar el primer anidamiento de bucles tiene los últimos elementos accedidos de los arrays



Al empezar el segundo anidamiento se acceden los primeros elementos de los arrays, que ya no están



La caché al finalizar el primer anidamiento de bucles tiene los últimos elementos accedidos de los arrays



Fusión de bucles: Solución

```
for( i=0; i<N; i++)  
    for( j=0; j<N; j++) {  
        a[i][j] = 1 / b[i][j] * c[i][j];  
        d[i][j] = a[i][j] + c[i][j];  
    }
```

- Los datos se tienen que cargar una única vez
- Además pueden reusarse en registros
 - En el ejemplo, $a[i][j]$ y $c[i][j]$
- Ahorro en instrucciones de control de bucles



Partición en bloques: Problema

```
float a[10], v[1000];  
for( i=0; i<10; i++)  
    for( j=0; j<1000; j++)  
        a[i] += i * v[j];
```

Caché de 256 bytes, líneas de 16 bytes, asociativa de dos vías

$i=0 \Rightarrow$ Se carga $a[0]$ y $v[0:999]$, total 4016 bytes, ~ 251 fallos

v[0]	v[1]	v[2]	v[3]	v[4]	v[5]	v[6]	v[7]	...	v[28]	v[29]	v[30]	v[31]
v[32]	v[33]	v[34]	v[35]	v[36]	v[37]	v[38]	v[39]		v[60]	v[61]	v[62]	v[63]

$i=1 \Rightarrow$ Se carga $a[1]$ y $v[0:999]$, total 4016 bytes, ~ 251 fallos

...

En total hay aproximadamente ~ 2510 fallos



Partición en bloques: Solución

```
float a[10], v[1000];  
for( jj=0; jj<1000; jj+=64)  
    for( i=0; i<10; i++)  
        for( j=jj; j<min(jj+64,1000); j++)  
            a[i] += i * v[j];
```

Caché de 256 bytes, líneas de 16 bytes, asociativa de dos vías

jj=0 $\Rightarrow$ i=0 $\Rightarrow$ Se carga a[0] y v[0:63], ~1+16 fallos
i=1 $\Rightarrow$ Se (re)carga a[1] y v[0:63], ~1+2 fallos
...
i=9 $\Rightarrow$ Se (re)carga a[9] y v[0:63], ~1+2 fallos } ~44 fallos

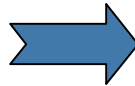
jj=1 $\Rightarrow$ i=0 $\Rightarrow$ Se carga a[0] y v[64:127], ~1+16 fallos
i=1 $\Rightarrow$ Se (re)carga a[1] y v[64:127], ~1+2 fallos
...
i=9 $\Rightarrow$ Se (re)carga a[9] y v[64:127], ~1+2 fallos } ~44 fallos

...
En total hay aproximadamente ~440 fallos



Partición en bloques: Uso de CPU

```
float a[10], v[1000];  
for( i=0; i<10; i++)  
    for( j=0; j<1000; j++)  
        a[i] += i * v[j];
```



```
float a[10], v[1000];  
for( jj=0; jj<1000; jj+=64)  
    for( i=0; i<10; i++)  
        for( j=jj; j<min(jj+64,1000); j++)  
            a[i] += i * v[j];
```

- 10 iteraciones del bucle i
- 10000 iteraciones del bucle j
- 11 predicciones erróneas de salto

- 16 iteraciones del bucle jj
- 160 iteraciones del bucle i
- 10000 iteraciones del bucle j
- 177 predicciones erróneas de salto
- Cálculos del límite del bucle j

El coste depende mucho del compilador y flags de optimización empleados

Partición en bloques: Ayudando a un mal compilador

```
float a[10], v[1000];  
for( jj=0; jj<1000; jj+=64)  
    for( i=0; i<10; i++)  
        for( j=jj; j<min(jj+64,1000); j++)  
            a[i] += i * v[j];
```



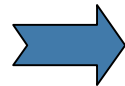
```
float a[10], v[1000];  
for( jj=0; jj<1000; jj+=64) {  
    const int limj = min(jj+64, 10000);  
    for( i=0; i<10; i++)  
        for( j=jj; j<limj; j++)  
            a[i] += i * v[j];  
}
```

Prebúsqueda

Idea: requerir que los datos se traigan a caché antes del momento en el que son necesarios. Permite solapar su acercamiento al procesador con computaciones precedentes

- Hardware: El programador no puede controlarla
- Software: Automática por el compilador o controlada por programador

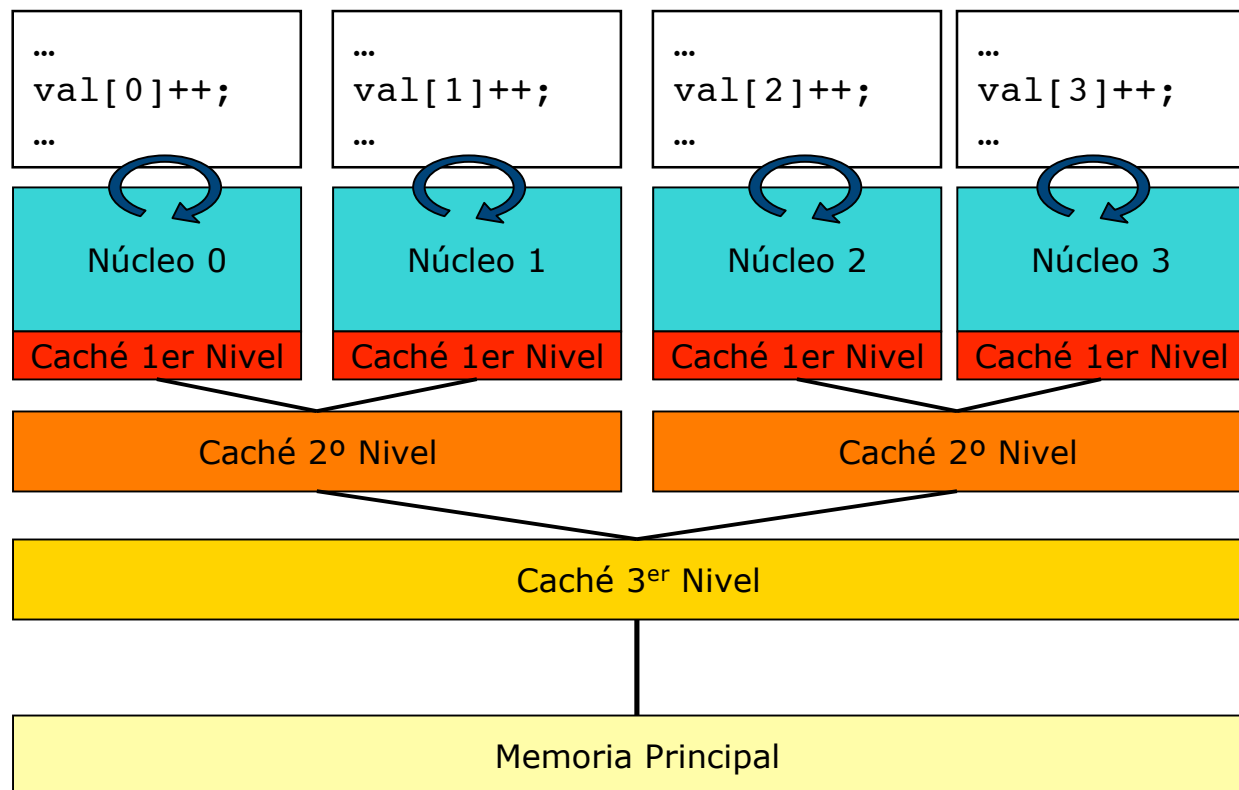
```
for( i= 0; i < 2000; i++)  
    t += x[i][i] + sqrt(y[i][i]);
```



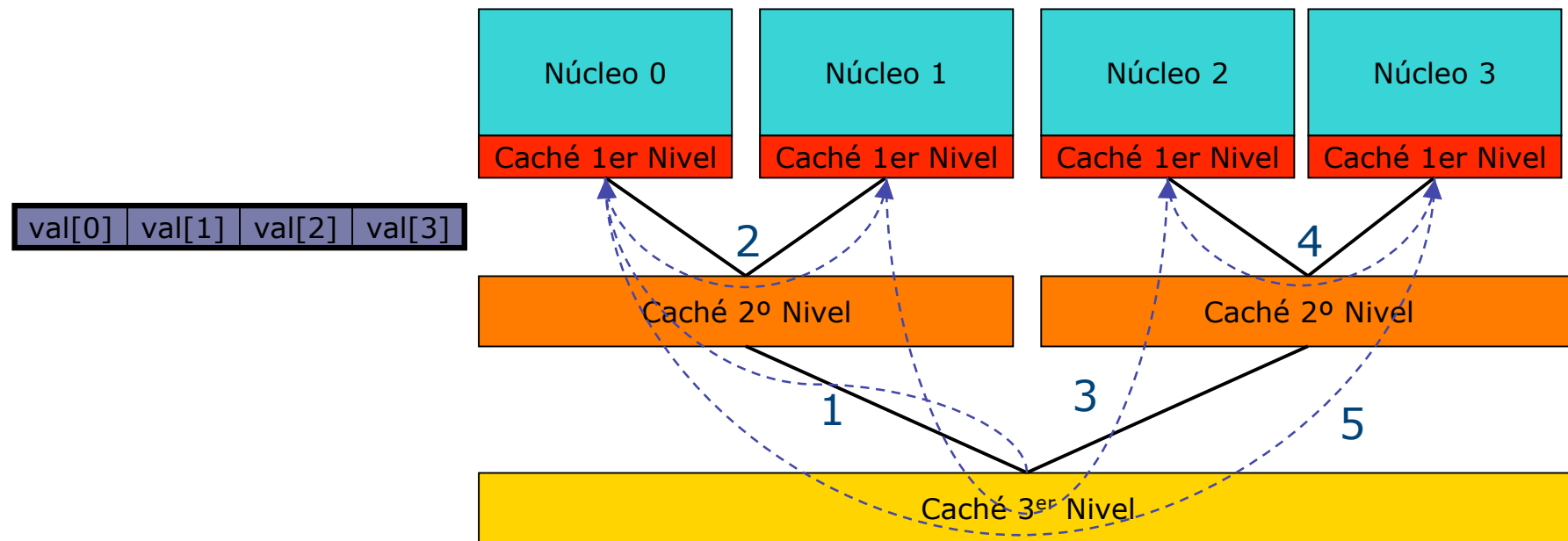
```
for( i= 0; i < 2000; i++) {  
    __builtin_prefetch(&x[i+8][i+8]);  
    __builtin_prefetch(&y[i+8][i+8]);  
    t += x[i][i] + sqrt(y[i][i]);  
}
```

- El exceso de prebúsquedas degrada el rendimiento
 - Supone ejecutar más instrucciones
 - Ej.: No hacer prebúsquedas consecutivas a la misma línea
- La prebúsqueda no debe hacerse ni demasiado pronto ni demasiado tarde

Multiprocesadores: Compartición falsa (I)



Multiprocesadores: Compartición falsa (II)

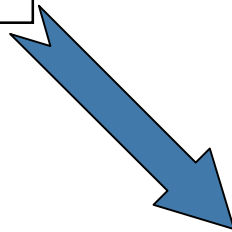


Compartición falsa: Solución

```
int mi_id;  
double val[nthreads];  
...  
void f() {  
    int i;  
    for(i=0; ...) {  
        val[mi_id] += ...;  
    }  
}
```



```
int mi_id;  
const int dist = LINE_SIZE/sizeof(double);  
double val[nthreads * dist];  
...  
void f() {  
    int i;  
    for(i=0; ...) {  
        val[mi_id * dist] += ...;  
    }  
}
```



```
int mi_id;  
double val[nthreads];  
...  
void f() {  
    int i;  
    double tmp = val[mi_id];  
    for(i=0; ...) {  
        tmp += ...;  
    }  
    val[mi_id] = tmp;  
}
```