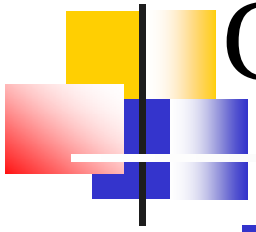
Programación Paralela y Computación de Altas Prestaciones
Algoritmos Matriciales por Bloques

Javier Cuenca

Dpto. de Ingeniería y Tecnología de Computadores

Universidad de Murcia



Contenido

- Trabajo por bloques
- Multiplicación de matrices
- Factorización LU
- Optimización automática: Tamaño de bloque óptimo

Trabajo por bloques

- En las operaciones anteriores los costes son:

	Coste Computacional	Memoria
Vector-vector	n	n
Matriz-vector	n^2	n^2
Matriz-matriz	n^3	n^2

- Desarrollando algoritmos con operaciones matriz-matriz, para el mismo número de operaciones aritméticas menos accesos a memoria → menor tiempo de ejecución

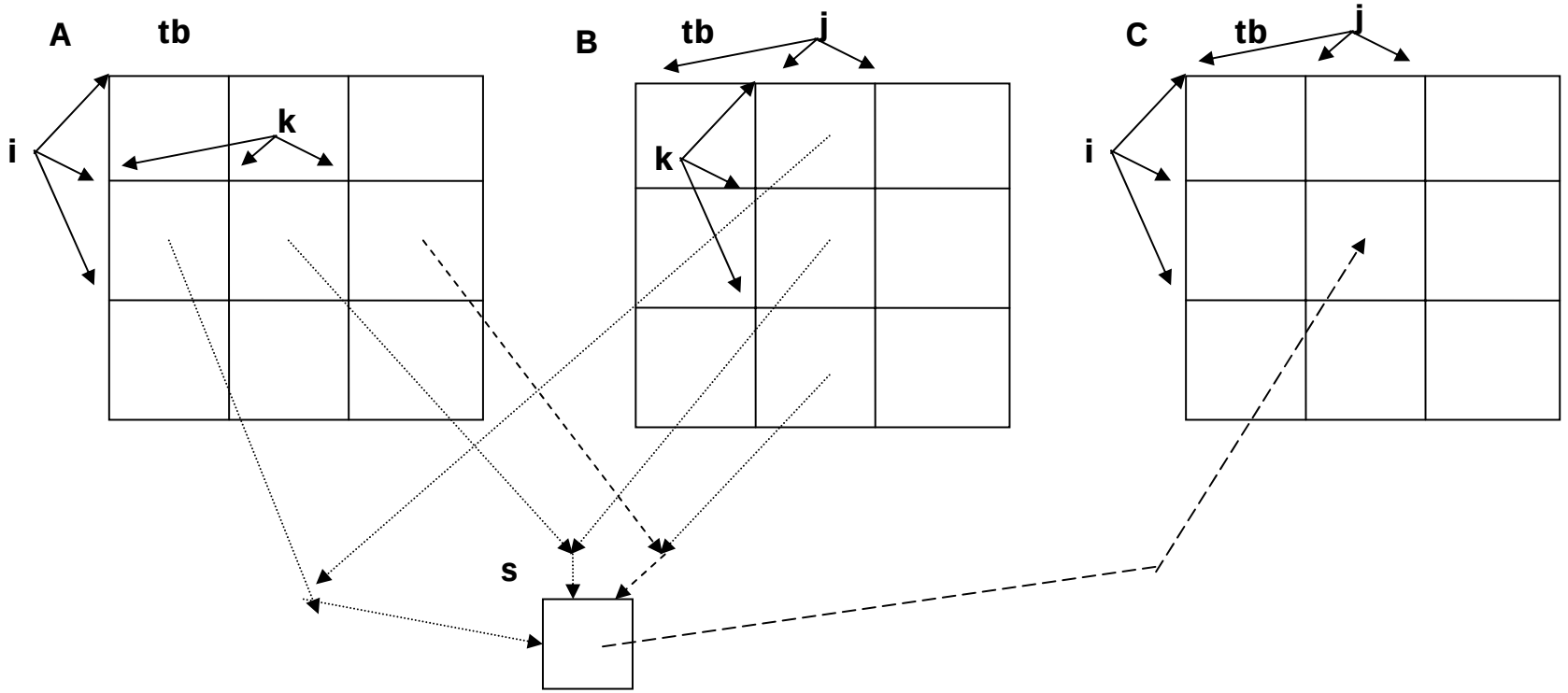
- Usado en el desarrollo de librerías desde los 80 (BLAS-3, LAPACK)

- Posibilidad de migración a rutinas paralelas más escalables

Trabajo por bloques

- La reducción varía de un sistema a otro
- ¿Cómo se sabe el tamaño de bloque óptimo? Varía con:
  - Sistema
  - Problema a resolver
  - Tamaño del problema
- Con lo que el método preferido también varía con el tamaño y el sistema *(polialgoritmos)*
- Difícil determinar a priori mejor método y parámetros → métodos de *optimización automática*

Multiplicación de matrices



Multiplicación de matrices

```
void matriz_matriz_ld (double *a,int fa,int ca,int lda,
    double *b,int fb,int cb,int ldb,double *c,int fc,int
      cc,int ldc)
{ int i,j,k;

  double s;
  for(i=0;i<fa;i++)
      for(j=0;j<cb;j++)
      {
          s=0.;
          for(k=0;k<ca;k++)
              s+=a[i*lda+k]*b[k*ldb+j];
          c[i*ldc+j]=s;
      }
```

Algoritmo sin bloques (normal). Acceso elemento a elemento.

Problemas pequeños: buenas prestaciones pues caben en memoria de niveles bajos de la jerarquía.

Problemas grandes: peores prestaciones.

Multiplicación de matrices

```
void matriz_matriz_bloques  (double *a,int fa,int ca,int lda,double
   *b,int fb,int cb,int ldb,double *c,int fc,int cc,int ldc,int tb)
{    int i,j,k;  double *s;

   s=(double *) malloc(sizeof(double)* tb *
   for(i=0;i<fa;i=i+ tb)
           for(j=0;j<cb;j=j+ tb)
           {
               matriz_cero(s, tb, tb, tb);
               for(k=0;k<ca;k=k+ tb)
                   multiplicar_acumular_matrices(&a[i*lda+k], tb,
                   tb,lda,&b[k*ldb+j],tb, tb,ldb,s, tb, tb, tb);

               copiar_matriz(s,tb, tb, tb,&c[i*ldc+j], tb, tb,ldc);
           }
   free(s);
}
```

Algoritmo por bloques.
Acceso y operaciones por bloques .
Buenas prestaciones independiente del tamaño.
El tamaño de bloque es parámetro a determinar.

```
void multiplicar_acumular_matrices(double *a,int fa,int ca,int lda,double *b,int
fb,int cb,int ldb,double *c,int fc,int cc,int ldc)
{
  int i,j,k,kb;
  double *da,*db,s;

  for(i=0;i<fa;i++)
  {
    da=&a[i*lda];
    for(j=0;j<cb;j++)
    {
      db=&b[j];
      s=c[i*ldc+j];
      for(k=0,kb=0;k<ca;k++,kb=kb+ldb)
      {
        s=s+da[k]*db[kb];
      }
      c[i*ldc+j]=s;
    }
  }
}
```

```
                    multiplicar_acumular_matrices(&a[i*lda+k], tb,
                    tb,lda,&b[k*ldb+j],tb, tb,ldb,s, tb, tb, tb);

                copiar_matriz(s,tb, tb, tb,&c[i*ldc+j], tb, tb,ldc);
            }
        free(s);
    }
```

Multiplicación de matrices

```
void matriz_matriz_bloquesdobles (double *a,int fa,int ca,int
   lda,double *b,int fb,int cb,int ldb,double *c,int fc,int cc,int
   ldc,int tb,int tbp)
{    int i,j,k; double *s;

   s=(double *) malloc(sizeof(double)* tb * t
   for(i=0;i<fa;i=i+ tb)
           for(j=0;j<cb;j=j+ tb)
           {
               matriz_cero(s, tb, tb, tb);
               for(k=0;k<ca;k=k+ tb)
                    multiplicar_acumular_bloques(&a[i*lda+k],tb,
                       tb,lda,&b[k*ldb+j],tb,tb,ldb,s,tb,tb, tb, tbp);
               copiar_matriz(s, tb, tb, tb,&c[i*ldc+j], tb, tb,ldc);
           }
   free(s);
}
```

Algoritmo por bloques dobles. La operación sobre bloques no es la multiplicación directa, sino por bloques. Tenemos dos tamaños de bloque.

```
void multiplicar_acumular_bloques(double *a,int fa,int ca,int lda,double *b,int fb,int cb,int
ldb,double *c,int fc,int cc,int ldc,int tbp)
{
  int i,j,k;
  double *s;

  s=(double *) malloc(sizeof(double)*tbp*tbp);

  for(i=0;i<fa;i=i+tbp)
  {
    for(j=0;j<cb;j=j+tbp)
    {
      copiar_matriz(&c[i*ldc+j],tbp,tbp,ldc,s,tbp,tbp,tbp);
      for(k=0;k<ca;k=k+tbp)
      {
        multiplicar_acumular_matrices(&a[i*lda+k],tbp,tbp,lda,&b[k*ldb+j],tbp,tbp,ldb,s,tbp,tbp,tbp
      }
      copiar_matriz(s,tbp,tbp,tbp,&c[i*ldc+j],tbp,tbp,ldc);
    }
  }
  free(s);
```

```
                {
                    matriz_cero(s, tb, tb, tb);
                    for(k=0;k<ca;k=k+ tb)
                        multiplicar_acumular_bloques(&a[i*lda+k],tb,
                          tb,lda,&b[k*ldb+j],tb,tb,ldb,s,tb,tb, tb, tbp);
                    copiar_matriz(s, tb, tb, tb,&c[i*ldc+j], tb, tb,ldc);
                }
      free(s);
   }
```

Multiplicación de matrices

Almacenamiento por bloques:

matriz

0	1	2	3
4	5	6	7
8	9	10	11
12	13	14	15

almacenamiento

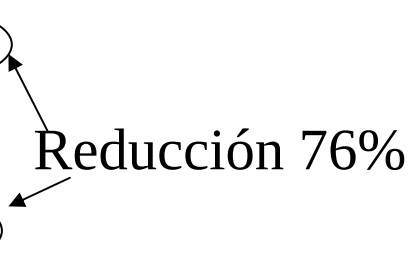
0	1	4	5
2	3	6	7
8	9	12	13
10	11	14	15

posible acceso más rápido a los datos dentro de las operaciones por bloques

Multiplicación de matrices

- Multiplicación de matrices, en portátil:

Método\tamaño		1000	1200	1400
normal		12.70	21.95	36.41
bloques	25	3.69	6.30	9.25
	50	3.56	5.90	8.71
	100	4.25	6.33	8.95
bloques dobles	25 5	4.67	7.87	10.89
	50 10	5.03	8.08	12.93
	50 25	4.53	7.16	11.11
	100 20	4.87	7.33	10.97
	100 25	4.78	7.06	9.92
	100 50	3.90	5.85	8.92



Multiplicación de matrices

- Multiplicación de matrices, en SUN Ultra 1:

Método\tamaño		200	400	800
Normal		**0.2179**	13.4601	217.5464
Traspuesta		**0.2013**	3.3653	27.9945
Bloques	**10**	0.2880	2.5901	21.9029
	25	0.2192	1.8347	14.9642
	50	**0.2161**	**1.7709**	**14.2502**
Bloq tras	**10**	0.2937	2.5026	20.4405
	25	0.2195	**1.8009**	14.6415
	50	**0.2152**	**1.7628**	**14.1806**
Almac blo	**10**	0.2949	2.5122	20.3762
	25	0.2277	1.8490	14.8625
	50	0.2296	1.8429	14.7314
Bl tr al bl	**10**	0.2925	2.4985	20.1975
	25	0.2244	1.8082	**14.5282**
	50	0.2231	**1.7147**	**13.6553**
Bloq dob	**20 5**	0.6105	4.9363	39.9594
	20 10	0.3206	2.6669	19.7044
	50 10	0.3039	2.4542	19.7044
	50 25	0.2370	1.9221	15.5190

Reducción 93%

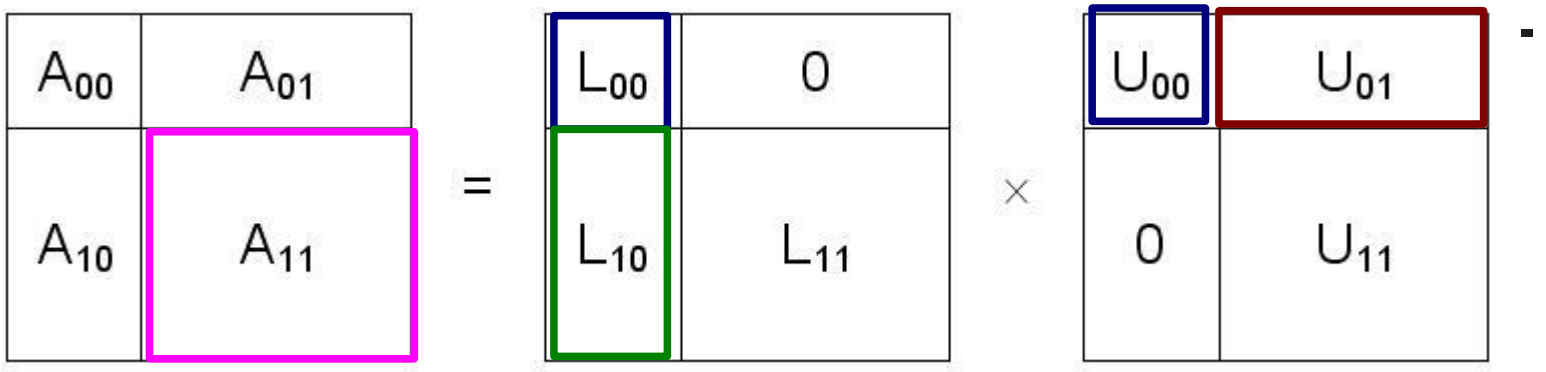
Multiplicación de matrices

- Multiplicación de matrices,en kefren, pentium 4:

Método\tamaño	200	400	800
Normal	0.0463	0.7854	7.9686
Traspuesta	**0.0231**	0.2875	2.3190
Bloques 10	**0.0255**	0.2493	**2.0327**
25	0.0265	**0.2033**	**1.6928**
50	**0.0219**	**0.1785**	**1.6594**
Bloq dob 20 5	0.0393	0.3669	3.4955
20 10	0.0269	0.3090	2.4424
50 10	0.0316	**0.2232**	2.2768
50 25	**0.0215**	**0.1755**	**1.4726**

Reducción 79%

Factorización LU



Cada A_{ij}, L_{ij}, U_{ij} de tamaño b×n :

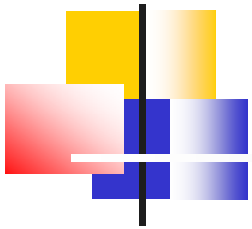
Paso 1: $L_{00}\,U_{00}=A_{00}$ →Factorización sin bloques

Paso 2: $L_{00}\,U_{01}=A_{01}$ → Sistema múltiple triangular inferior (¿bloques?)

Paso 3: $L_{10}\,U_{00}=A_{10}$ → Sistema múltiple triangular superior (¿bloques?)

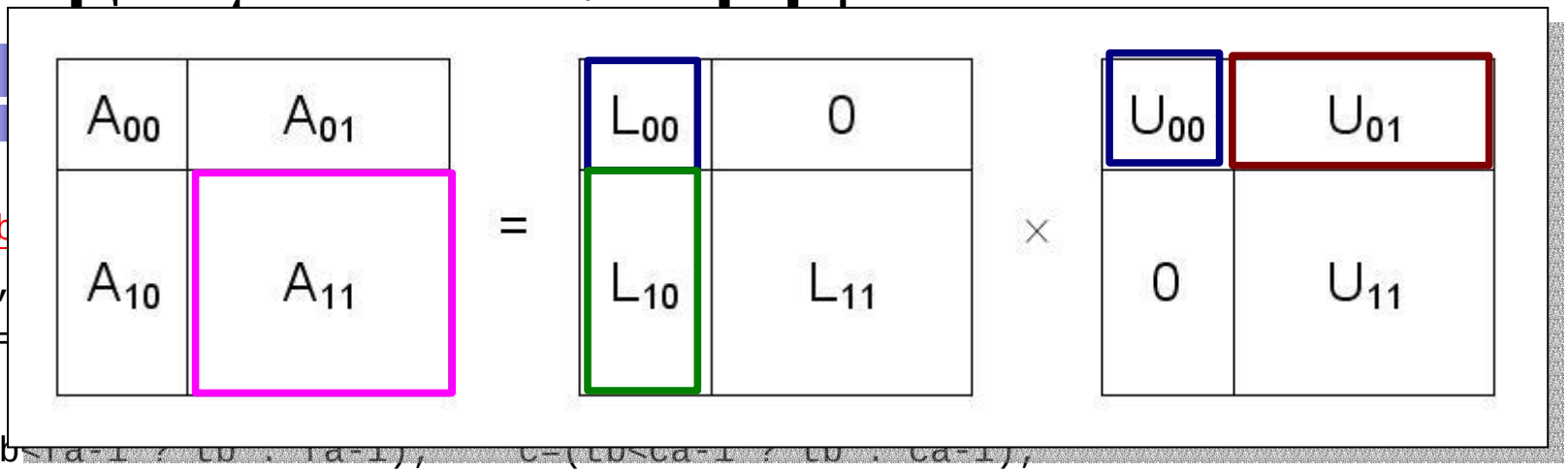
Paso 4: $A_{11}=L_{10}\,U_{01}+L_{11}\,U_{11}$ → $A'_{11}=A_{11}-L_{10}\,U_{01}$, por bloques

y seguir trabajando con el nuevo valor de A_{11}

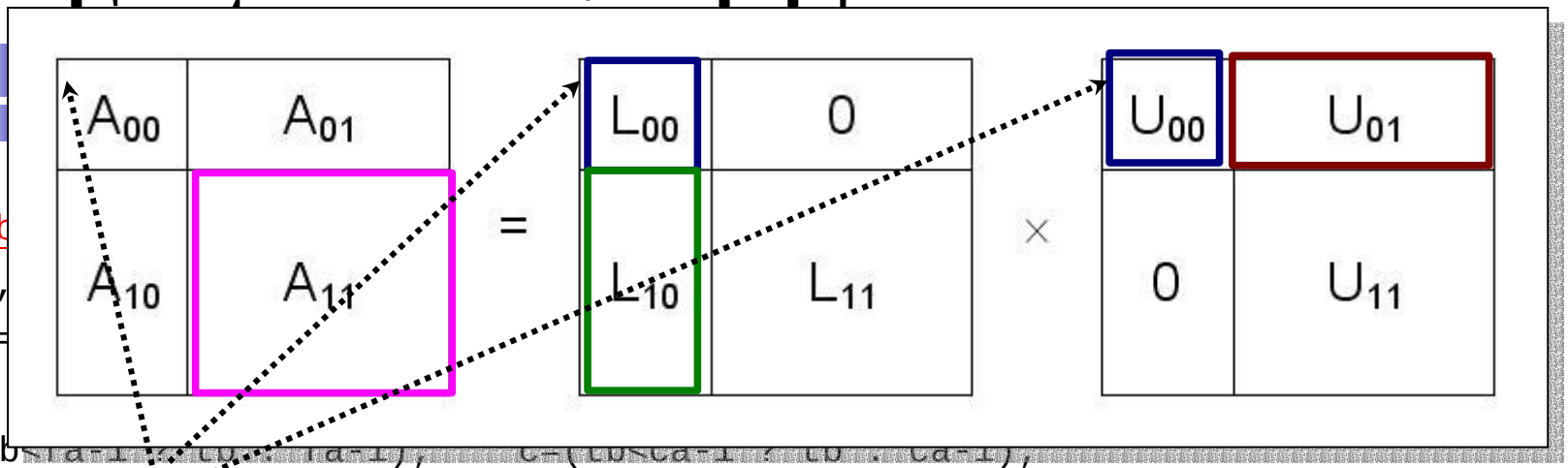


Factorización LU

```
void lu_bloques (double *a,int fa,int ca,int lda,int tb)
{int i,j,k,f,c;
   for(i=0;i<fa;i=i+tb)
   {
    f=(tb<fa-i ? tb : fa-i);    c=(tb<ca-i ? tb : ca-i);
    lu(&a[i*lda+i],f,c,lda);      //1
    if(i+tb<fa)
    {
        sistema_triangular_inferior(&a[i*lda+i],f,c,lda,&a[i*lda+i+c],f,ca-i-c,lda);  //2

        sistema_triangular_superior(&a[i*lda+i],f,c,lda,&a[(i+f)*lda+i], fa-i-f, c,lda);//3

        multiplicar_restar_matrices(&a[(i+f)*lda+i],fa-i-f,c,lda,

      &a[i*lda+i+f],f,ca-i-c,lda,&a[(i+f)*lda+i+c],fa-i-f,ca-i-c,lda); //4
    } } }
```

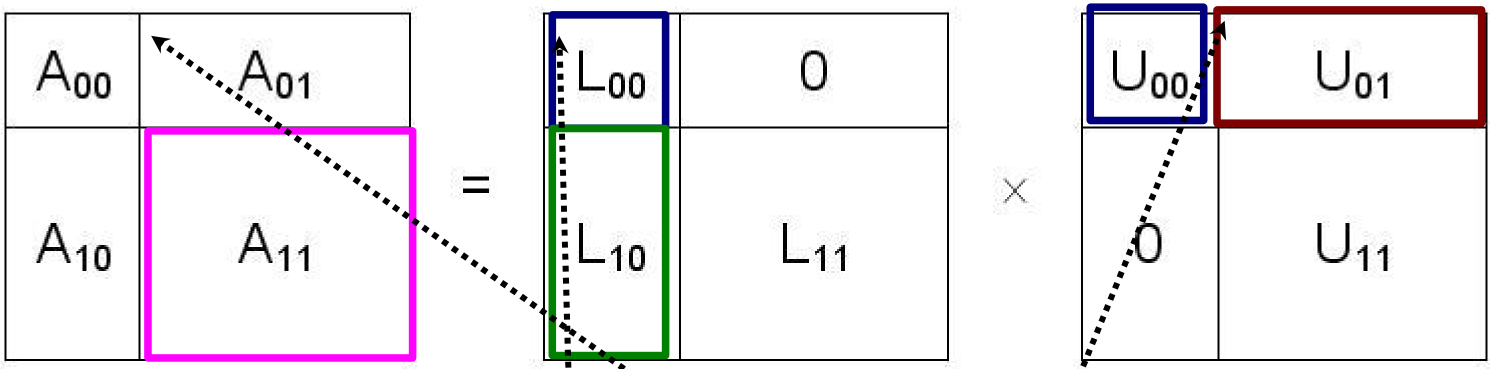



```
void lu_b
{int i,j,
   for(i=
   {
    f=(tb<fa-i ? tb : fa-i);    c=(tb<ca-i ? tb : ca-i);
    lu(&a[i*lda+i],f,c,lda);      //1
    if(i+tb<fa)
    {
        sistema_triangular_inferior(&a[i*lda+i],f,c,lda,&a[i*lda+i+c],f,ca-i-c,lda);  //2

        sistema_triangular_superior(&a[i*lda+i],f,c,lda,&a[(i+f)*lda+i], fa-i-f, c,lda);//3

        multiplicar_restar_matrices(&a[(i+f)*lda+i],fa-i-f,c,lda,

      &a[i*lda+i+f],f,ca-i-c,lda,&a[(i+f)*lda+i+c],fa-i-f,ca-i-c,lda); //4
    } } }
```



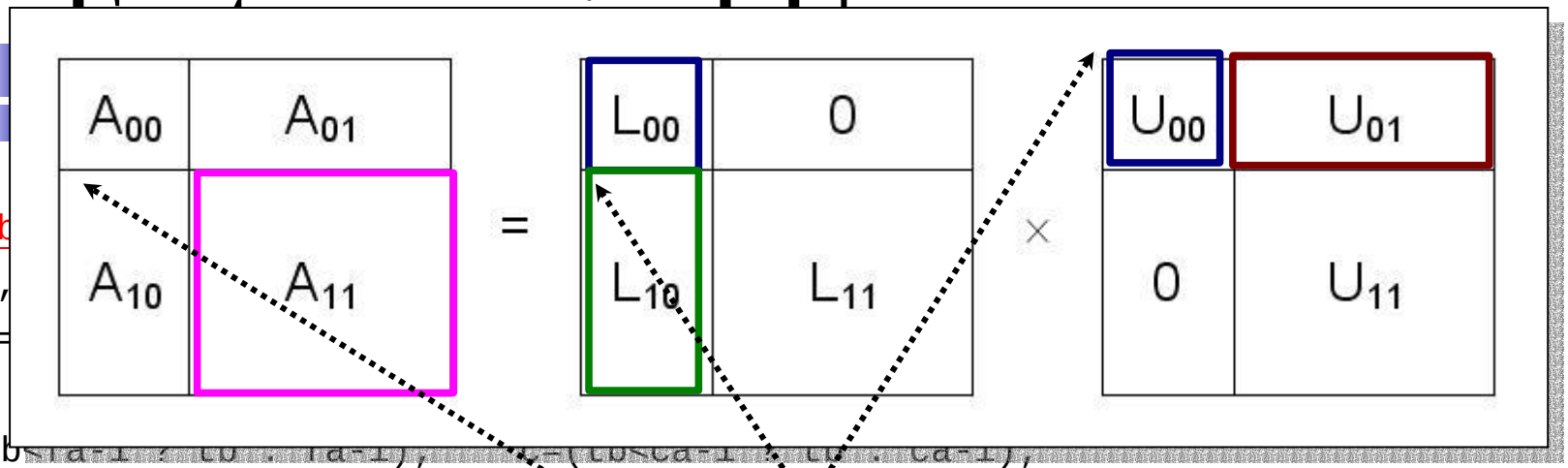

```
void lu_b
{int i,j,
   for(i=
   {
    f=(tb<fa-i ? tb : fa-i);    c=(tb<ca-i ? tb : ca-i);
    lu(&a[i*lda+i],f,c,lda);      //1
    if(i+tb<fa)
    {
        sistema_triangular_inferior(&a[i*lda+i],f,c,lda,&a[i*lda+i+c],f,ca-i-c,lda);  //2

                                                                                    );//3
        &
   } }
```

Paso 1: $L_{00}\,U_{00}=A_{00}$ →Factorización sin bloques



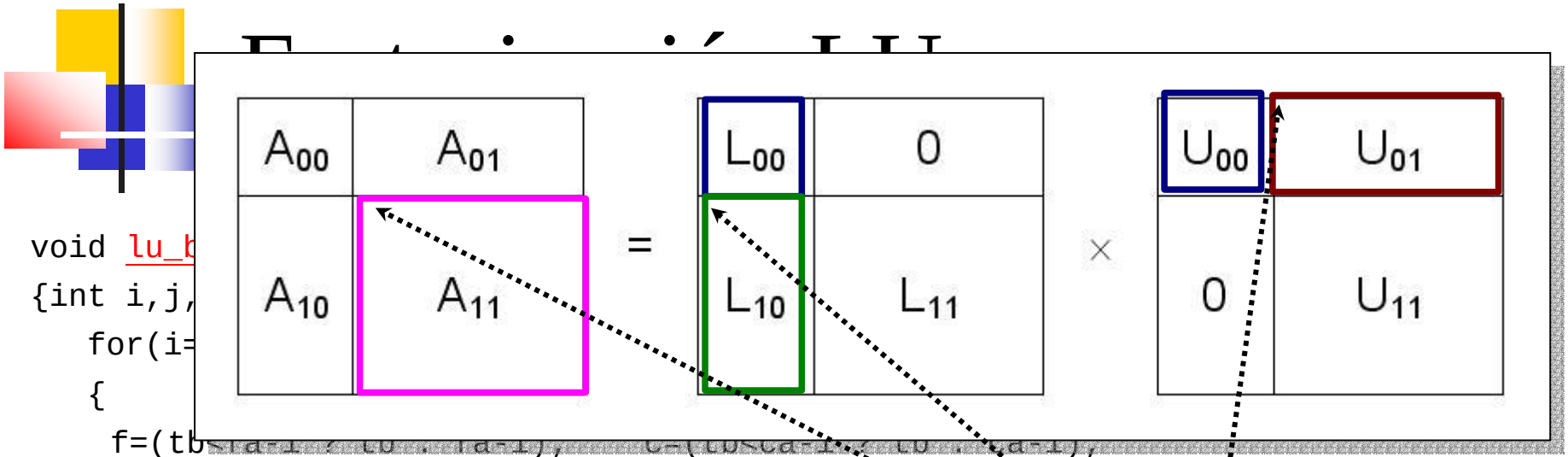

```
void lu_b
{int i,j,
    for(i=
    {
     f=(tb
     lu(&a[i*lda+i],f,c,lda);      //1
     if(i+tb<fa)
     {
        sistema_triangular_inferior(&a[i*lda+i],f,c,lda,&a[i*lda+i+c],f,ca-i-c,lda);  //2
                                                                              );//3
       &
     } }
```

Paso 2: $L_{00} U_{01} = A_{01}$ → Sistema múltiple triangular inferior




```
void lu_b
{int i,j,
   for(i=
   {
    f=(tb<fa-i ? tb : fa-i);    c=(tb<ca-i ? tb : ca-i);
    lu(&a[i*lda+i],f,c,lda);      //1
    if(i+tb<fa)
    {
        sistema_triangular_inferior(&a[i*lda+i],f,c,lda,&a[i*lda+i+c],f,ca-i-c,lda);  //2

        sistema_triangular_superior(&a[i*lda+i],f,c,lda,&a[(i+f)*lda+i], fa-i-f, c,lda);//3

      &
   } }
```

Paso 3: $L_{10}\, U_{00} = A_{10}$ → Sistema múltiple triangular superior




```
void lu_b
{int i,j,
   for(i=
   {
    f=(tb<fa-1 ? tb : fa-1);    c=(tb<ca-1? tb : ca-1);
    lu(
    if(
    {
                                                          //2
      sistema_triangular_superior(&a[i*lda+i],f,c,lda,&a[(i+f)*lda+i], fa-i-f, c,lda);//3

      multiplicar_restar_matrices(&a[(i+f)*lda+i],fa-i-f,c,lda,

   &a[i*lda+i+f],f,ca-i-c,lda,&a[(i+f)*lda+i+c],fa-i-f,ca-i-c,lda); //4
   } } }
```

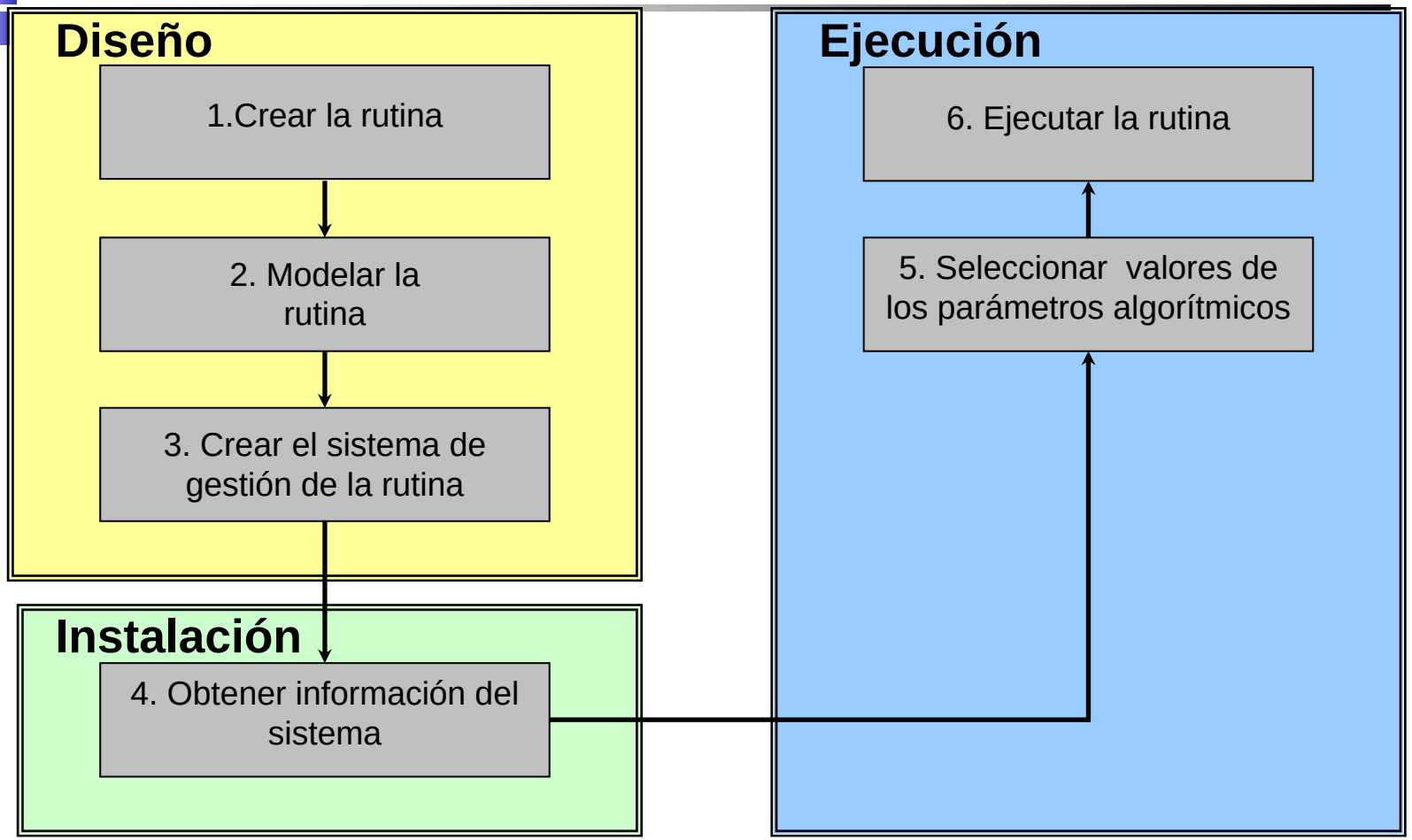
Paso 4: $A_{11} = L_{10} U_{01} + L_{11} U_{11} \rightarrow A'_{11} = A_{11} - L_{10} U_{01}$

Factorización LU

en mi portátil:

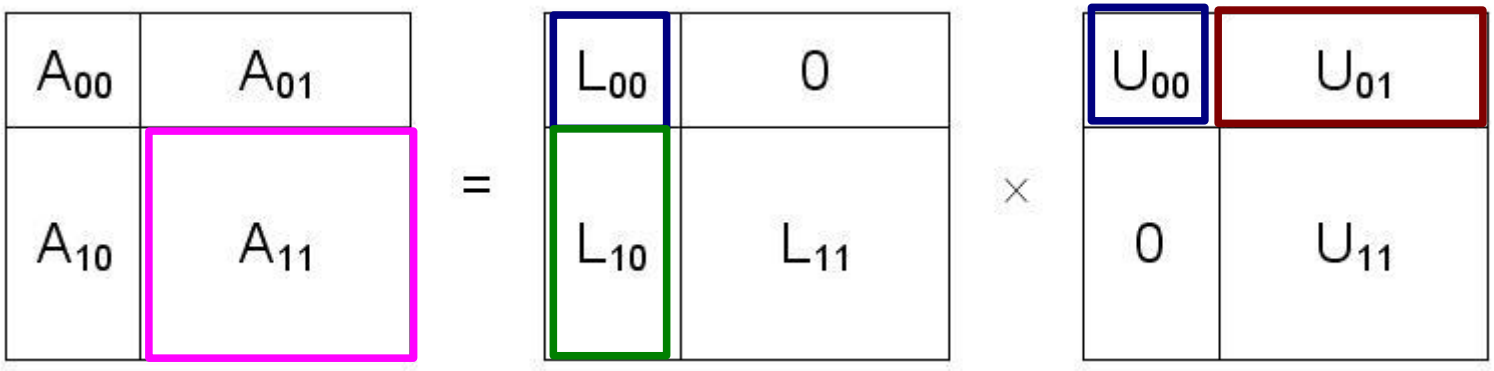
tamaño bloque\matriz	800	1000
1	2.10	4.01
12	1.42	2.78
25	1.29	2.27
37	1.24	2.37
44	1.20	2.00
50	1.22	2.32
100	1.47	2.24
200	2.29	3.47
400	2.17	3.67
sin bloques	1.73	3.43

Optimización Automática:
Tamaño de bloque óptimo



Optimización Automática.
Ejemplo: Factorización LU

1. Crear la rutina



DGETF2: Paso 1: $L_{00}\,U_{00}=A_{00}$ → Factorización sin bloques

DTRSM: Paso 2: $L_{00}\,U_{01}=A_{01}$ → Sistema múltiple triangular inferior

DTRSM: Paso 3: $L_{10}\,U_{00}=A_{10}$ → Sistema múltiple triangular superior

DGEMM: Paso 4: $A_{11}=L_{10}\,U_{01}+L_{11}\,U_{11}$ → $A'_{11}=A_{11}-L_{10}\,U_{01}$

Optimización Automática.
Ejemplo: Factorización LU

2. Modelar el tiempo de ejecución:

$$T_{EXEC} = \frac{2}{3} n^3 k_{3\_DGEMM} + bn^2 k_{3\_DTRSM} + \frac{1}{3} b^2 n k_{2\_DGETF2}$$

- ***n***: el tamaño del problema a resolver
- ***SP***: parámetros del sistema
  - k_{3_DGEMM}, k_{3_DTRSM}, k_{2_DGETF2}
  - coste computacional de una operación básica realizada por rutinas utilizadas (*DGEMM*, *DTRSM*, *DGETF2*)
- ***AP***: parámetros algorítmicos
  - ***b***: tamaño de bloque

Optimización Automática.
Ejemplo: Factorización LU

4. Obtener información del sistema

Table 1. Estimation of SP (k_3) for different block sizes and systems (in microseconds).

System		Block size 16	32	64	96	128
SUN1	ref-BLAS	0.0205	**0.0195**	0.0220	0.0240	0.0340
	mac-BLAS	0.0120	**0.0100**	**0.0100**	**0.0100**	**0.0100**
	ATLAS	0.0070	0.0060	**0.0055**	**0.0055**	**0.0055**
SUN5	ref-BLAS	0.0135	0.0130	**0.0128**	**0.0128**	**0.0128**
	mac-BLAS	0.0060	0.0050	**0.0044**	**0.0044**	**0.0044**
	ATLAS	0.0040	0.0032	0.0030	**0.0028**	**0.0028**
PPC	mac-BLAS	0.0025	0.0022	0.0020	**0.0019**	**0.0019**
R10K	mac-BLAS	0.0105	0.0035	0.0030	**0.0028**	**0.0028**

Optimización Automática.
Ejemplo: Factorización LU

5. Seleccionar valores de los *AP*

Table 1. Estimation of SP (k_3) for different block sizes and systems (in microseconds).

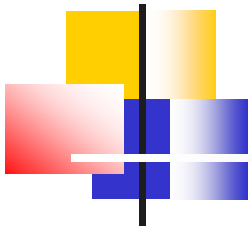
System		Block size 16	32	64	96	128
SUN1	ref-BLAS	0.0205	**0.0195**	0.0220	0.0240	0.0340
	mac-BLAS	0.0120	**0.0100**	**0.0100**	**0.0100**	**0.0100**
	ATLAS	0.0070	0.0060	**0.0055**	**0.0055**	**0.0055**
SUN5	ref-BLAS	0.0135	0.0130	**0.0128**	**0.0128**	**0.0128**
	mac-BLAS	0.0060	0.0050	**0.0044**	**0.0044**	**0.0044**
	ATLAS	0.0040	0.0032	0.0030	**0.0028**	**0.0028**
PPC	mac-BLAS	0.0025	0.0022	0.0020	**0.0019**	**0.0019**
R10K	mac-BLAS	0.0105	0.0035	0.0030	**0.0028**	**0.0028**

Optimización Automática.
Ejemplo: Factorización LU

6. Ejecución de la rutina

Table 2. Comparison of the optimum execution time (opt), the execution time with the AP (block size) chosen by the model (OAP), the weighted execution time (wei), and LAPACK, on R10K.

	Execution time				Deviation from the Optimum		
n	**opt**	**OAP**	**wei**	**LAPACK**	**OAP**	**wei**	**LAPACK**
512	0.27	0.29	0.43	0.31	7%	59%	15%
1024	2.57	2.60	3.70	2.71	1%	44%	5%
1536	6.52	6.52	11.00	7.26	0%	69%	11%
2048	40.71	41.56	46.60	43.97	2%	14%	8%
2560	29.90	29.90	51.38	31.04	0%	72%	4%
3072	59.87	60.52	93.82	65.96	1%	57%	10%



Trabajo alumnos.

- Conectarse a `luna.inf.um.es`
- Copiar a tu directorio los ejemplos que están en: `/home/javiercm/ejemplos_algmatblo`
- Probar los programas de las sesiones y corregir errores
- Comparar los tiempos de las multiplicaciones matriciales (bloques y no bloques)
- Comparar los tiempos de la factorización LU (bloques y no bloques)
- En las multiplicaciones matriciales por bloques quitar la restricción de:

  tamaño de las matrices = multiplo del tamaño de bloque