Computación de Altas Prestaciones

Evaluación Algoritmos a bloques, Valgrind

Analisis del producto de matrices , orden j, k, i

```
For j=1:n
        For k=1:n
                For i=1:n
                        C(i,j)=C(i,j)+A(i,k)*B(k,j)
                End
        End
End
```

Desde el punto de vista del número de accesos a memoria, la matriz A se carga entera en cada iteración del bucle j.

Convendría poder reusar mas los datos, de forma que no se tuviera que cargar cada dato n veces.

Para lograr esto, se utilizan los algoritmos por bloques.

Objetivo de los algoritmos a bloques

Los algoritmos a bloques son variaciones de algoritmos tradicionales, en los que un problema grande se descompone en trozos más pequeños que “caben” en la memoria cache. Se usan para problemas “Grandes”.

Seleccionando cuidadosamente el tamaño de los bloques y el algoritmo, se puede minimizar el tráfico de memoria necesario para resolver el problema.

Todas las implementaciones en librerías de algebra lineal numérica (solución de Sistemas de Ecuaciones Lineales, cálculo de valores y vectores propios, solución de problemas de mínimos cuadrados, etc.) están hechas con Algoritmos a bloques, basados en el producto de matrices.

Algoritmo base:

En la sesión de teoría, mostramos como era una versión de producto de matrices a bloques en lenguaje Matlab:

```
For i=1:N       { N es el número de bloques}
  ind_i=(i-1)*tb+1:i*tb
  For j=1:N
     ind_j=(j-1)*tb+1:j*tb
        For k=1:N
           ind_k=(k-1)*tb+1:k*tb
           C(ind_i,ind_j)=A(ind_i,ind_k)*B(ind_k,ind_j)+C(ind_i,ind_j)
        End
  End
End
```

Detalles sobre este algorimo

-Suponemos que las tres matrices son cuadradas y del mismo tamaño, N*tb

-Como el tamaño de la matriz no tiene porque ser múltiplo del tamaño de bloque, no es un algoritmo “realista”: habría que completarlo con código para manejar los “bordes” de la matriz.

-El tamaño de bloque debe ser algo “pequeño” para que los bloques quepan en la cache, pero , por eficiencia, lo mas grande posible. Vamos a intentar hallarlo para este problema experimentando.

-En poliformat hay una programa en C ya escrito con tres versiones del producto de matrices, vamos a utilizar ese programa para experimentar, pero antes vamos a ver como son esas tres versiones

Algoritmo sin bloques

La primera versión contra la que vamos a comparar es la versión “normal”, sin bloques, ordenación j k i, suponiendo ordenación por columnas y la forma de referenciar matrices en C que vimos en el seminario 2.

```
void matprod(double *A, double *B, double *C, int n)
{    int i,j,k;
     for (j=0; j<n; j++)
        for(k=0;k<n; k++)
           for(i=0;i<n; i++)
              C[i+j*n]= C[i+j*n]+ A[i+k*n]* B[k+j*n];
}
```

Algoritmo a bloques, versión 1

La segunda versión sería una “traducción” del algoritmo de la transparencia: es decir, como vamos a ver, Necesitamos la “Función externa” con un triple bucle “a nivel de bloques” y la función interna para multiplicar bloques, con un triple bucle a nivel de elementos de matrices. La función interna será parecida al producto sin bloques:

Versión inicial

```
void matprod_peq1(double *A, double *B, double *C, int n)
{    int i,j,k;
     for (j=0; j<n; j++)
        for(k=0;k<n; k++)
           for(i=0;i<n; i++)
              C[i+j*n]= C[i+j*n]+ A[i+k*n]* B[k+j*n];
}
```

Algoritmo a bloques, versión 1

La versión inicial “no vale” porque multiplica matrices “enteras”:
Hay que pasar como parámetros los índices de bucles correspondientes, para poder localizar las posiciones correctas: pasamos i, j, k como parámetros.

También el tamaño de bloque tb , para poder calcular correctamente los rangos de filas

Algoritmo a bloques, versión 1

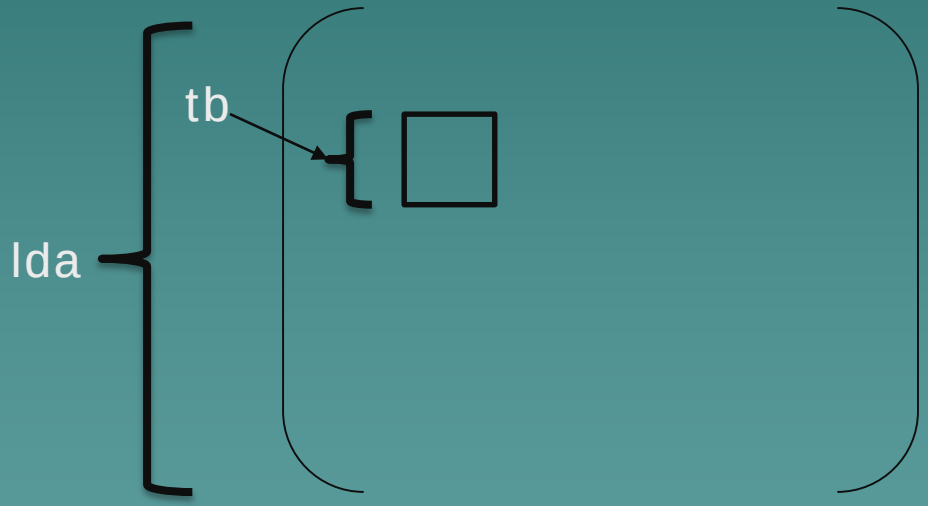
Función interna versión 2 =>

```
void matblockpeq2(double *A, double *B, double *C, int ib, int jb, int kb,
int tb)
{
   int i,j,k;
   int inii=ib*tb;
   int inij=jb*tb;
   int inik=kb*tb;
   for (j=inij; j<inij+tb; j++)
      for(k=inik;k<inik+tb; k++)
         for(i=inii;i<inii+tb; i++)
       C[i+j*tb]= C[i+j*tb]+ A[i+k*tb]* B[k+j*tb];
}
```

Ojo , todavía no es correcto!

Algoritmo a bloques, versión 1

Cuando trabajamos con bloques de una matriz, almacenada por columnas (por filas sería similar) Hay dos dimensiones involucradas: La dimensión del bloque (tb) y la dimensión de la matriz que la contiene (mas exactamente, el número de filas) A este número se le suele llamar leading dimensión o lda



Para referenciar correctamente los elementos dentro de la matriz necesitamos el leading dimension

Algoritmo a bloques para producto de matrices cuadradas-1

Función interna correcta=>

```
void matblockpeq(double *A, double *B, double *C, int ib, int jb,
    int kb, int tb, int lda )
{
    int i,j,k;
    int inii=ib*tb;
    int inij=jb*tb;
    int inik=kb*tb;
    for (j=inij; j<inij+tb; j++)
        for(k=inik;k<inik+tb; k++)
            for(i=inii;i<inii+tb; i++)
         C[i+j*lda]= C[i+j*lda]+ A[i+k*lda]* B[k+j*lda];
}
```

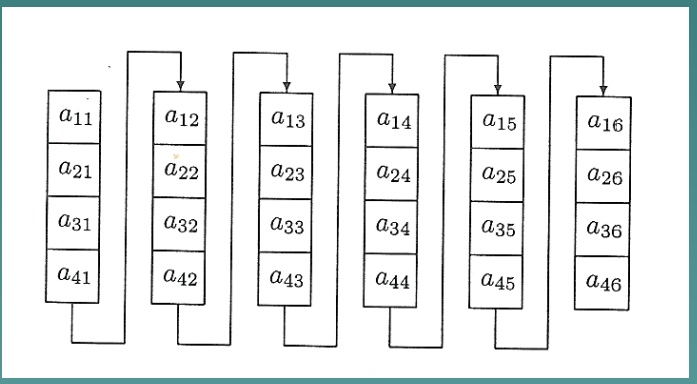
Algoritmo a bloques para producto de matrices cuadradas-1

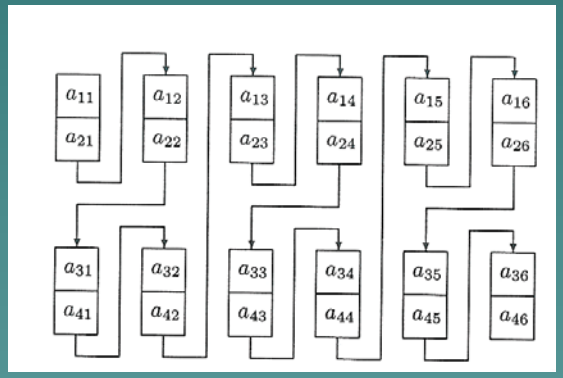
Función externa =>

```
void matprodbloques(double *A, double *B, double *C, int lda, int
    tb )
{    int i,j,k;
     int nb=lda/tb;
     for (j=0; j<nb; j++)
          for(k=0;k<nb; k++)
            for(i=0;i<nb; i++)
              matblockpeq(A,B,C,i,j,k,tb,lda);
}
```

Versión 2: Algoritmos a bloques “con copia”

En muchos casos, antes de aplicar el algoritmo se copia la matriz a formato “a bloques”.

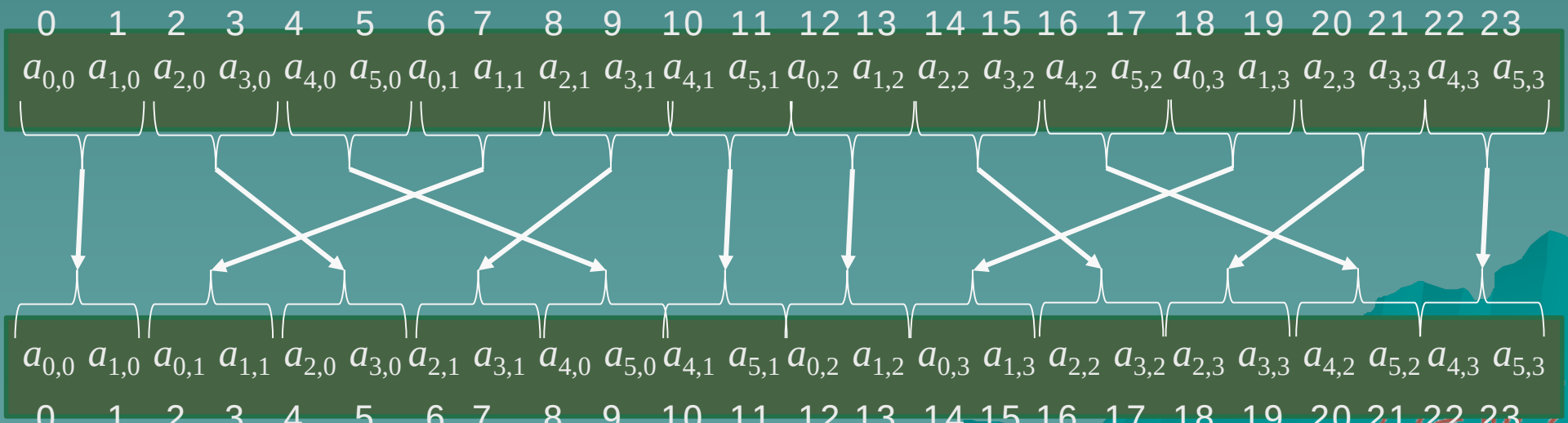




Esto se hace dependiendo del problema a resolver y de la dimensión de las matrices (según si el tiempo gastado en la copia se va a amortizar o no)

Estructura en memoria

$$A = \begin{pmatrix} a_{0,0} & a_{0,1} & a_{0,2} & a_{0,3} \\ a_{1,0} & a_{1,1} & a_{1,2} & a_{1,3} \\ a_{2,0} & a_{2,1} & a_{2,2} & a_{2,3} \\ a_{3,0} & a_{3,1} & a_{3,2} & a_{3,3} \\ a_{4,0} & a_{4,1} & a_{4,2} & a_{4,3} \\ a_{5,0} & a_{5,1} & a_{5,2} & a_{5,3} \end{pmatrix}$$



Producto de matrices bloques “con copia”

Para hacer una comparación justa, tomaremos tiempo tanto de la copia de A y B a formato de bloques como de la copia “de vuelta “ de C a formato normal.

Necesitamos subrutinas para copiar matrices a formato de bloques: To_Blocked, From_Blocked

Desventaja 1 de versión con copia: La copia de las matrices lleva su tiempo. Para matrices pequeñas no vale la pena.

Ventaja 1: La función interna de la versión con copia no necesita lda (solo hay una dimensión involucrada), y (si el tamaño de bloque se elige correctamente), todo el bloque cabe en la cache y habrá menos fallos de cache

Algoritmo a bloques “con copia”

Subrutina interna para el caso “con copia”:

```
void Mult_add(double *A, double *B, double *C,int i_bar, int
  j_bar, int k_bar, int n_bar, int tb)
{
  int b_sqr=tb*tb;
  double *c_p = C+(i_bar + j_bar*n_bar)*b_sqr;;
  double *a_p = A + (i_bar + k_bar*n_bar)*b_sqr;
  double *b_p = B + (k_bar+ j_bar*n_bar )*b_sqr;
  int i, j, k;
  for (j = 0; j < tb; j++)
     for (k = 0; k < tb; k++)
        for (i = 0; i < tb; i++)
           c_p[i+j*tb] += a_p[i+k*tb]*(b_p[k+j*tb]);
}
```

Algoritmo a bloques “con copia”

Subrutina externa para el caso “con copia”:

```
void Blocked_mat_mult(double *A, double *B, double *C,  int
    n_bar,int tb)
{
   int i_bar, j_bar, k_bar;

   for (j_bar = 0; j_bar < n_bar; j_bar++)
      for (k_bar = 0; k_bar < n_bar; k_bar++) {
         for (i_bar = 0; i_bar < n_bar; i_bar++)
           Mult_add(A, B, C, i_bar, j_bar, k_bar,n_bar,tb);
      }
}  /* Blocked_mat_mult */
```

Experimentos (1)

La subrutina test_bloques_col.c permite experimentar con los tres algoritmos; el tamaño de bloque (tb) debe ser divisor exacto de la dimensión de las matrices (n).

Experimento 1, comprobar que el producto de matrices se hace correctamente (seleccionar un tamaño pequeño y descomentar las llamadas a la subrutina para escribir la matriz C)

Para los siguientes experimentos, comentaremos de nuevo la subrutina de escritura y seleccionaremos un tamaño grande (1600) y probaremos con diferentes tamaños:

Experimento 2: compilar con icc (o gcc) y –O3,

buscar mejor tamaño de bloques entre estos.

(8, 10, 16, 20, 25, 32, 50, 64, 100)

Experimentos (2)

Experimento 3, Probar los flags –vec-report2, -guide –parallel, -parallel, autooptimización (-prof-gen, -prof-use) combinados con –O3 (solo para icc)

Recuerda que con –parallel el compilador de intel usa su propia subrutina de producto de matrices, se podía comprobar usando gprof.

Valgrind

Valgrind se puede describir como un simulador de CPU en software.

-Su uso principal es el poder detectar errores de memoria típicos de C /C++ (acceso a posiciones de memoria ilegales, memory leaks, ...)

-Se han desarrollado otras herramientas que funcionan bajo valgrind: Cachegrind (cache profiler), HellGrind (Thread debugger), y otros.

www.valgrind.org

Experimentos (3): Estudio de fallos de cache usando valgrind.

Seleccionar un tamaño de matriz n no demasiado grande (no mas de 1000)

Compilar con icc, con –g y con –O3.

Ejecutar el programa “bajo valgrind”:

```
>> valgrind –- tool=cachegrind  nombre_ejecutable
```

Para poder estudiar los accesos a cache de cada uno de los tres algoritmos por separado, comentar las partes apropiadas del programa, recompilar, para ver los fallos de cache.