Capitolo 5

Alcune librerie per il calcolo matriciale numerico

La risoluzione computazionale della maggior parte dei problemi scientifici comporta l'esecuzione di operazioni di calcolo matriciale, cioè operazioni su vettori e matrici. È stato stimato che il calcolo per la risoluzione di un sistema lineare costituisce un nucleo del processo di risoluzione di circa il 70% dei problemi scientifici.

Poiché lo sviluppo di software efficiente deve tenere conto di fattori diversi, molti dei quali dipendono strettamente dall'ambiente di calcolo [36], è fondamentale disporre di una implementazione dei nuclei computazionali di base ottimizzata rispetto alla piattaforma di calcolo (software low-level e medium-level)[17]. Una organizzazione modulare del software costituisce la condizione necessaria per garantire al software applicativo le medesime caratteristiche (software medium-level e high-level).

In quest'ottica, a partire dalla fine degli anni '70, sono state sviluppate librerie di software per la risoluzione di problemi di calcolo matriciale.

Tali librerie forniscono i *building blocks* con i quali è possibile implementare algoritmi efficienti per la risoluzione di problemi piú

complessi.

Una delle difficoltà maggiori nella progettazione di software per architetture avanzate è rappresentata dalle prestazioni diverse delle unità funzionali di un calcolatore: se da un lato la tecnologia ha consentito la progettazione di processori in grado di raggiungere una peak performance di oltre 1 Tflop/s, dall'altro le prestazioni della memoria non sono aumentate proporzionalmente. Idealmente vorremmo che la memoria dei calcolatori fosse al contempo capace e veloce, ma i costi proibitivi costringono ad un compromesso. Per questo motivo la memoria viene progettata secondo una struttura gerarchica. Un utilizzo ottimale della memoria si ottiene progettando algoritmi che riusino i dati trasferiti nella memoria *"veloce"* minimizzando gli spostamenti tra i vari livelli. Per comprendere le implicazioni derivanti da ciò, consideriamo un semplice esempio [6]: *supponiamo che la memoria abbia solo 2 livelli, uno lento ed uno veloce.*

In una situazione ideale, tutti i dati si trovano nella memoria veloce ed il tempo di esecuzione di un algoritmo è

$$T = f \cdot t_{calc}$$

dove f è il numero di operazioni floating-point eseguite, t_{calc} il tempo di esecuzione di una singola operazione.

In generale, invece, nella valutazione del tempo T dobbiamo tener conto anche del tempo di trasferimento dei dati dalla memoria *"lenta"* alla memoria *"veloce"*. Pertanto, detto m il numero di elementi trasferiti dalla memoria veloce alla lenta e viceversa e t_m il tempo impiegato per ognuno di tali trasferimenti, nel nostro modello semplificato, risulta:

$$T = f \cdot t_{calc} + m \cdot t_m$$

dove, tipicamente, $t_m \gg t_{calc}$. Se poniamo

$$q = \frac{f}{m}$$

e riscriviamo

$$T = f \cdot t_{calc} \left(1 + \frac{1}{q} \cdot \frac{t_m}{t_{calc}}\right)$$

appare evidente che ci avviciniamo alla situazione ideale quanto piú elevato è il valore di q.

In generale si cerca di stabilire il peso di $m \cdot t_m$ rispetto a $f \cdot t_{calc}$, cioè si calcola

$$\frac{m \cdot t_m}{f \cdot t_{calc}} = \lambda\mu$$

dove

$$\lambda = \frac{t_m}{t_{calc}}$$

è un parametro caratteristico della macchina e

$$\mu = \frac{m}{f} = \frac{1}{q}$$

è caratteristico dell'algoritmo. Il parametro μ può essere visto come una misura del *"traffico parassita"* di dati. L'obiettivo nel disegno degli algoritmi è quindi minimizzare μ.

Queste considerazioni sono alla base della formulazione *Block-partitioned* e, piú in generale, a blocchi, degli algoritmi del calcolo matriciale numerico: l'idea è quella di sostituire le operazioni scalari con le corrispondenti operazioni vettoriali, in modo da aumentare il rapporto tra il numero di operazioni effettuate e gli accessi ai livelli *"lenti"* della memoria.

Con questi obiettivi è nato il package **BLAS** (**B**asic **L**iear **A**lgebra

Subprograms). Il package è sviluppato su tre livelli, BLAS1 (1974) [34], BLAS2 (1984) [9], BLAS3 (1987) [10]: il livello 1 opera su coppie di vettori, il livello 2 effettua operazioni di tipo matrice-vettore e il livello 3 lavora su coppie di matrici.

Nel paragrafo successivo vediamo come e con che finalità è stato sviluppato BLAS considerando la risoluzione di un sistema di equazioni con il metodo di eliminazione di Gauss.

5.1 Rivisitazione dell'algoritmo di eliminazione di Gauss: da BLAS1 a BLAS3

Problema

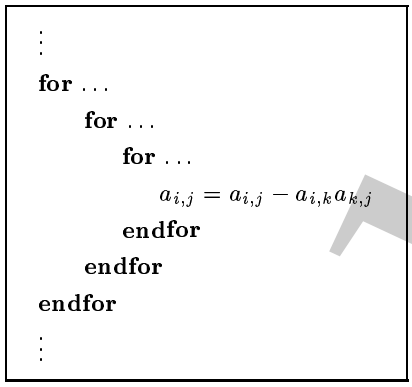
Risoluzione del sistema

$$Ax = b$$

con $A \in \Re^{n \times n}$, x *e* $b \in \Re^n$ *e* A *matrice non singolare, mediante l'algoritmo di eliminazione di Gauss.*

L'algoritmo di eliminazione di Gauss[1] risulta composto di tre cicli *for ... endfor* (*Procedura* 5.1) innestati, di cui il primo indica i passi, gli altri due operano rispettivamente su righe e colonne della matrice *"attiva"*. Indichiamo rispettivamente con k, i e j i cicli sui passi, sulle righe e sulle colonne della matrice.

[1] In questo caso non si fa riferimento all'algoritmo di Gauss con pivoting parziale. Il nostro obiettivo, infatti, è studiare l'efficienza, trascurando per il momento le questioni legate alla stabilità.



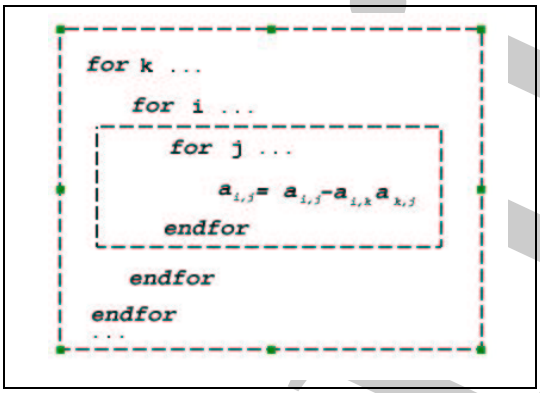
Procedura 5.1 - Schema per l'algoritmo di eliminazione di Gauss.

L'ordine dei tre cicli sugli indici i, j e k non è stato ancora fissato. Indipendentemente dall'ordine dei tre cicli la complessità asintotica di tempo è sempre $T = O(n^3)$. Al variare dell'ordine dei cicli, invece, può cambiare in modo significativo il tempo di esecuzione del software che implementa l'algoritmo in quanto quest'ultimo dipende anche dagli accessi in memoria. Se, ad esempio, il ciclo sulle righe, i, è esterno al ciclo sulle colonne, j, gli accessi in memoria agli elementi della matrice sono effettuati per righe. Infatti, fissata la riga $i - esima$, al variare di j vengono prelevati tutti gli elementi della fissata riga. Invertendo i due cicli, invece, viene fissata la colonna $j - esima$, l'indice i scorre gli elementi della colonna $j - esima$. Gli accessi in memoria agli elementi della matrice sono dunque effettuati per colonne.

Se, ad esempio, si utilizza come linguaggio di programmazione il Fortran, è utile tenere presente che esso memorizza le matrici per colonne. Conviene dunque utilizzare la versione dell'algoritmo di Gauss con il ciclo sull'indice j esterno al ciclo sull'indice i.

Esaminiamo ora i nuclei computazionali di base nell'algoritmo di Gauss nella versione dell'algoritmo che prevede il ciclo sui passi k, poi il ciclo i sulle righe ed infine sulle colonne, j, come mostrato

nella *Procedura* 5.2 .



Procedura 5.2 - Schema dell'algoritmo di eliminazione di Gauss nella versione kij.

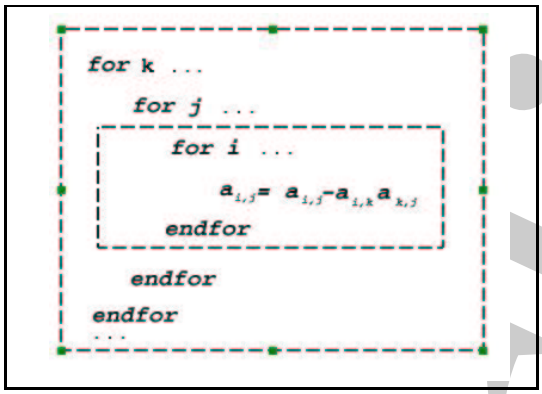
Fissati gli indici k ed i, l'operazione di base, evidenziata nella *Procedura* 5.2 , nel riquadro interno tratteggiato, è **l'aggiornamento di un vettore riga**, $\{a_{i,j}\}_j$, mediante il prodotto di uno scalare, $a_{i,k}$, per un vettore riga, $\{a_{k,j}\}_j$. Infatti, fissati gli indici k ed i, l'elemento $a_{i,k}$ è uno scalare, mentre gli elementi $a_{i,j}$ e $a_{k,j}$ al variare dell'indice j sono vettori riga. In generale si ha un'operazione del tipo $y \leftarrow y + \alpha x$ con α scalare e x, y vettori riga (saxpy)[2].

Se invece si invertono i cicli e si considerano gli aggiornamenti nell'ordine k, j ed i, tenuti fissi k e j l'operazione di base è **l'aggiornamento di un vettore colonna**, $\{a_{i,j}\}_i$, mediante il prodotto di uno scalare, $a_{k,j}$, per un vettore colonna, $\{a_{i,k}\}_i$. Infatti, fissati gli indici k e j l'elemento $a_{k,j}$ è uno scalare, mentre gli elementi $a_{i,j}$ e $a_{i,k}$, al variare dell'indice i sono vettori colonna. L'algoritmo è mostrato nella *Procedura* 5.3 . Anche in questo caso si ha un'operazione del tipo $y \leftarrow y + \alpha x$ con α scalare e x, y vettori colonna.

[2] *saxpy* è un acronimo dell'operazione **s**calar **a**lpha per **x** **p**lus **y** ovvero:

$$y \leftarrow y + \alpha x$$

In BLAS1 la *s* iniziale sta per singola precisione, ma ci sono anche *daxpy*, in doppia precisione, *caxpy*, per dati complessi e *zaxpy*, per dati complessi in doppia precisione.



Procedura 5.3 - Schema dell'algoritmo di eliminazione di Gauss nella versione kji.

Pertanto, sia nella versione kij che in quella kji, come mostrato in Fig. 5.1, l'operazione di base è del tipo *saxpy*:

$$y \leftarrow y + \alpha x$$

cosí specificata nel livello I di BLAS:

$$saxpy(n,alpha,x,incx,y,incy)$$

I parametri $incx$ ed $incy$ rappresentano i passi con cui si considerano gli elementi dei vettori x e y rispettivamente. La complessità asintotica di tempo della saxpy è $T = O(n)$.

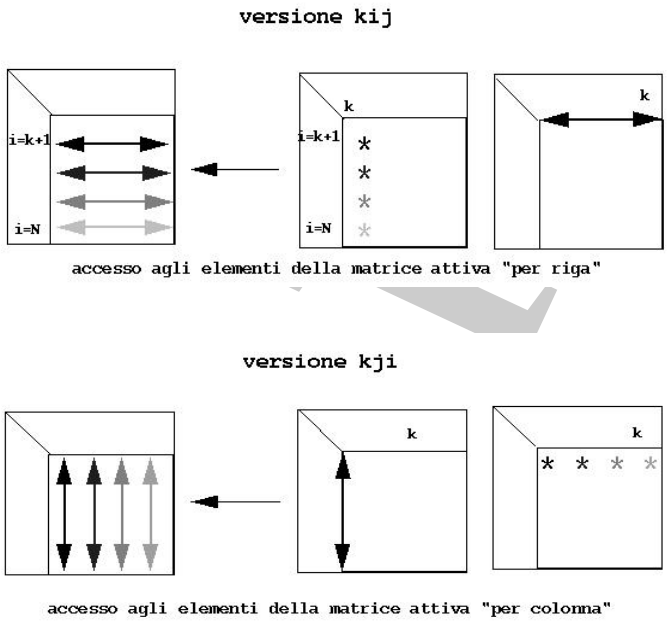


Figura 5.1: *In figura sono riportate le due combinazioni di indici, kij e kji. Gli asterischi rappresentano i singoli elementi della matrice. Con le frecce orizzontali vengono indicate le righe della matrice, mentre con le frecce verticali vengono rappresentate le colonne della matrice. Nella versione kij le righe della matrice sono calcolate mediante modifiche successive. Ogni modifica è un prodotto tra gli elementi di una colonna e una fissata riga. Nella versione kji le colonne della matrice sono calcolate mediante modifiche successive. Ogni modifica è un prodotto tra una fissata colonna e gli elementi di una riga.*

```
      subroutine saxpy(n,sa,sx,incx,sy,incy)
c
c     constant times a vector plus a vector.
c     uses unrolled loop for increments equal to one.
c     jack dongarra, linpack, 3/11/78.
c     modified 12/3/93, array(1) declarations changed to array(*)
c
      real sx(*),sy(*),sa
      integer i,incx,incy,ix,iy,m,mp1,n
c
      if(n.le.0)return
      if (sa .eq. 0.0) return
      if(incx.eq.1.and.incy.eq.1)go to 20
c
c        code for unequal increments or equal increments
c          not equal to 1
c
      ix = 1
      iy = 1
      if(incx.lt.0)ix = (-n+1)*incx + 1
      if(incy.lt.0)iy = (-n+1)*incy + 1
      do 10 i = 1,n
        sy(iy) = sy(iy) + sa*sx(ix)
        ix = ix + incx
        iy = iy + incy
   10 continue
      return
c        code for both increments equal to 1
c        clean-up loop
c
   20 m = mod(n,4)
      if( m .eq. 0 ) go to 40
      do 30 i = 1,m
        sy(i) = sy(i) + sa*sx(i)
   30 continue
      if( n .lt. 4 ) return
   40 mp1 = m + 1
      do 50 i = mp1,n,4
        sy(i) = sy(i) + sa*sx(i)
        sy(i + 1) = sy(i + 1) + sa*sx(i + 1)
        sy(i + 2) = sy(i + 2) + sa*sx(i + 2)
        sy(i + 3) = sy(i + 3) + sa*sx(i + 3)
   50 continue
      return
      end
```

Procedura 5.4 - Subroutine saxpy del livello 1 di BLAS.

Il livello 1 di BLAS è basato su operazioni del tipo $y \leftarrow y + \alpha x$.

Vediamo come viene eseguita l'operazione $y \leftarrow y + \alpha x$.

Inizialmente y, x ed α vengono prelevati dai registri e riposti nell'unità funzionale pipelined. Qui viene effettuata l'operazione e il risultato viene riposto nuovamente nei registri. Il procedimento è

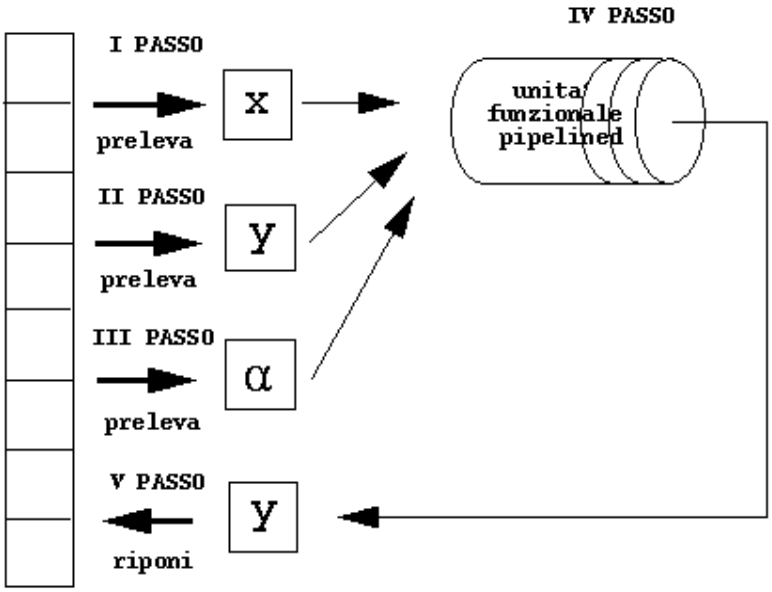


Figura 5.2: *L'operazione $y \leftarrow y + \alpha x$. Vengono prelevati dai registri i vettori x ed y e lo scalare α. Successivamente l'unità funzionale effettua l'operazione e il risultato y viene riposto in memoria.*

mostrato in Fig. 5.2.
Quindi, vengono effettuate le operazioni seguenti:

1. si preleva y; l'operazione richiede n accessi;
2. si preleva x; l'operazione richiede n accessi;
3. si preleva α; l'operazione richiede 1 accesso;
4. si esegue $y \leftarrow y + \alpha x$; l'operazione richiede $2n$ operazioni floating point;
5. si ripone y nel registro; l'operazione richiede n accessi.

Quindi per la *saxpy* risulta

$$\mu_{saxpy} = \frac{3n+1}{2n} = \frac{\#accessi}{\#op.fl.p.}$$

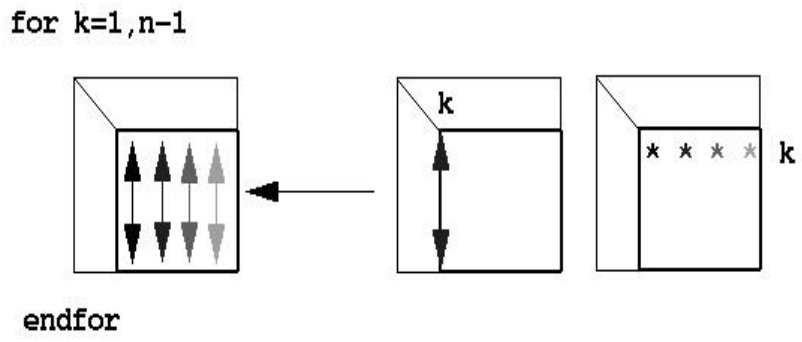


Figura 5.3: *Nel caso della combinazione kji ad ogni passo k viene calcolata una colonna della nuova matrice mediante il prodotto di una colonna (ottenuta fissando k e facendo variare i) per uno scalare (ottenuto fissando k e j). Ad ogni passo k la colonna $a_{i,k}$ viene prelevata e riposta in memoria j volte.*

Per n sufficientemente grande risulta

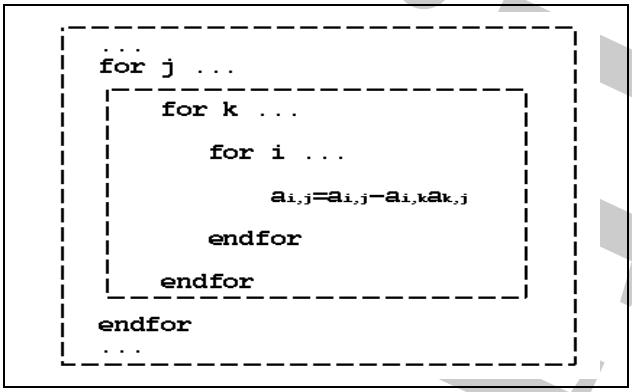
$$\mu_{saxpy} \approx \frac{3}{2}$$

Il numero di accessi ai registri è dunque maggiore del numero di operazioni floating point e la routine non è efficiente in termini del parametro μ, ovvero del traffico parassita dei dati.
Ricordiamo però che il livello 1 di BLAS è nato con altre motivazioni [34]. Gli obiettivi erano di fornire un prodotto software modulare, leggibile e standard. Nell'ottica di ridurre il parametro μ è stato infatti sviluppato un secondo livello di BLAS.

Riprendiamo l'algoritmo di eliminazione di Gauss nella versione kji. Notiamo che ogni colonna a^j viene aggiornata, cioè prelevata e riposta nel registro, j volte. In pratica non c'è riutilizzo dei vettori memorizzati nei registri. Infatti, fissato k, la colonna $\{a_{i,k}\}_i$ al variare di i viene prelevata per ogni j (Fig. 5.3).

Un modo piú efficiente di utilizzare i registri consiste nel mantenere la colonna a^j nei registri finchè l'aggiornamento non sia completato. A tal fine è opportuno riorganizzare i cicli dell'algoritmo

come illustrato nella *Procedura* 5.5, ovvero nell'ordine jki.



Procedura 5.5 - Schema dell'algoritmo di eliminazione di Gauss nella versione jki.

Fissato j, l'operazione di base, messa in risalto nel riquadro interno tratteggiato della *Procedura* 5.5 , è il prodotto di una matrice, $M = \{a_{i,k}\}$ al variare di k ed i, per un vettore colonna, $\{a_{k,j}\}$ al variare di k, ovvero è un'operazione del tipo:

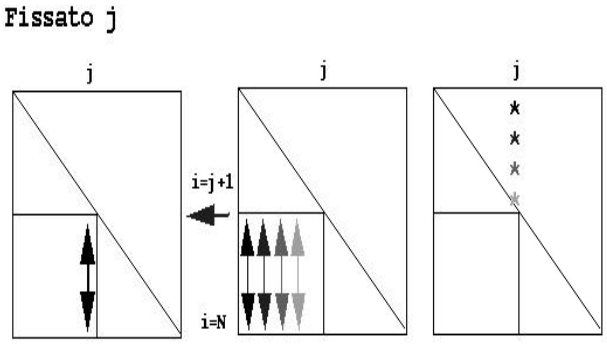
$$y \leftarrow y + Mx \quad x, y \in \Re^n, M \in \Re^{n \times n} \quad (gaxpy)$$

il vettore y viene aggiornato mediante un prodotto matrice per vettore.

```
for k = 1 to n do
    for i = 1 to n do
        y_i = y_i + m_ik x_k
    endfor
endfor
```

Procedura 5.6 - Schema dell'algoritmo della gaxpy.

Graficamente:



In tal caso la generica colonna j è calcolata definitivamente prima di essere riposta nella memoria centrale, mediante il prodotto di una matrice per la corrispondente colonna j.

Questa operazione richiede un numero di accessi ai registri inferiore rispetto alla *saxpy*. Infatti vengono eseguite le operazioni seguenti:

1. si preleva y; l'operazione richiede n accessi;
2. si preleva M; l'operazione richiede n^2 accessi;
3. si preleva x; l'operazione richiede n accessi;
4. si esegue $y \leftarrow y + Mx$; l'operazione richiede n^2 operazioni floating point;
5. si ripone y nel registro; l'operazione richiede n accessi.

In definitiva risulta

$$\mu_{gaxpy} = \frac{3n + n^2}{2n^2} = \frac{1}{2} + \frac{3}{2n}$$

Per n sufficientemente grande quindi

$$\mu_{gaxpy} \approx \frac{1}{2} < \frac{3}{2} \approx \mu_{saxpy}$$

Questo principio è alla base del livello 2 di BLAS.

I principali moduli di BLAS2 sono i seguenti:

- $y \leftarrow \alpha Ax + \beta y$ *matrice generica,* operazione eseguita dalla subroutine *sgemv(trans,n,alpha,a,lda,c,incx,beta,y,incy)*
- $y \leftarrow \alpha Ax + \beta y$ *matrice simmetrica,* operazione eseguita dalla subroutine *ssymv(uplo,n,alpha,a,x,incx,beta,y,incy)*
- $A \leftarrow \alpha xy^T + A$, operazione eseguita dalla subroutine *sger(m,n,alpha,a,x,incx,y,incy,lda)*

```
      SUBROUTINE SGEMV ( TRANS, M, N, ALPHA, A, LDA, X, INCX,
     $                   BETA, Y, INCY )
*     .. Scalar Arguments ..
      REAL               ALPHA, BETA
      INTEGER            INCX, INCY, LDA, M, N
      CHARACTER*1        TRANS
*     .. Array Arguments ..
      REAL               A( LDA, * ), X( * ), Y( * )
*
*  Purpose
*  =======
*  SGEMV  performs one of the matrix-vector operations
*
*     y := alpha*A*x + beta*y,   or   y := alpha*A'*x + beta*y,
*
*  where alpha and beta are scalars, x and y are vectors and A is an
*  m by n matrix.
*
*  Parameters
*  ==========
*
*  TRANS  - CHARACTER*1.
*           On entry, TRANS specifies the operation to be performed as
*           follows:
*
*              TRANS = 'N' or 'n'   y := alpha*A*x + beta*y.
*
*              TRANS = 'T' or 't'   y := alpha*A'*x + beta*y.
*
*              TRANS = 'C' or 'c'   y := alpha*A'*x + beta*y.
*           Unchanged on exit.
*  M      - INTEGER.
*           On entry, M specifies the number of rows of the matrix A.
*           M must be at least zero.
*           Unchanged on exit.
*
```

Procedura 5.7 - Routine SGEMV del livello 2 di BLAS - continua.

```
*
*  N      - INTEGER.
*           On entry, N specifies the number of columns of the matrix A.
*           N must be at least zero.
*           Unchanged on exit.
*
*  ALPHA  - REAL            .
*           On entry, ALPHA specifies the scalar alpha.
*           Unchanged on exit.
*
*  A      - REAL            array of DIMENSION ( LDA, n ).
*           Before entry, the leading m by n part of the array A must
*           contain the matrix of coefficients.
*           Unchanged on exit.
*
*  LDA    - INTEGER.
*           On entry, LDA specifies the first dimension of A as declared
*           in the calling (sub) program. LDA must be at least
*           max( 1, m ).
*           Unchanged on exit.
*
*  X      - REAL            array of DIMENSION at least
*           ( 1 + ( n - 1 )*abs( INCX ) ) when TRANS = 'N' or 'n'
*           and at least
*           ( 1 + ( m - 1 )*abs( INCX ) ) otherwise.
*           Before entry, the incremented array X must contain the
*           vector x.
*           Unchanged on exit.
*
*  INCX   - INTEGER.
*           On entry, INCX specifies the increment for the elements of
*           X. INCX must not be zero.
*           Unchanged on exit.
*
*  BETA   - REAL            .
*           On entry, BETA specifies the scalar beta. When BETA is
*           supplied as zero then Y need not be set on input.
*           Unchanged on exit.
*
*  Y      - REAL            array of DIMENSION at least
*           ( 1 + ( m - 1 )*abs( INCY ) ) when TRANS = 'N' or 'n'
*           and at least
*           ( 1 + ( n - 1 )*abs( INCY ) ) otherwise.
*           Before entry with BETA non-zero, the incremented array Y
*           must contain the vector y. On exit, Y is overwritten by the
*           updated vector y.
*
*  INCY   - INTEGER.
*           On entry, INCY specifies the increment for the elements of
*           Y. INCY must not be zero.
*           Unchanged on exit.
*
*  Level 2 Blas routine.
*
*  -- Written on 22-October-1986.
*     Jack Dongarra, Argonne National Lab.
*     Jeremy Du Croz, Nag Central Office.
*     Sven Hammarling, Nag Central Office.
*     Richard Hanson, Sandia National Labs.
*
*     .. Parameters ..
      REAL               ONE         , ZERO
      PARAMETER          ( ONE = 1.0E+0, ZERO = 0.0E+0 )
*     .. Local Scalars ..
      REAL               TEMP
```

Procedura 5.7 - Routine SGEMV del livello 2 di BLAS - continua.

```
      INTEGER            I, INFO, IX, IY, J, JX, JY, KX, KY, LENX, LENY
*     .. External Functions ..
      LOGICAL            LSAME
      EXTERNAL           LSAME
*     .. External Subroutines ..
      EXTERNAL           XERBLA
*     .. Intrinsic Functions ..
      INTRINSIC          MAX
*     .. Executable Statements ..
*     Test the input parameters.
*
      INFO = 0
      IF     ( .NOT.LSAME( TRANS, 'N' ).AND.
     $         .NOT.LSAME( TRANS, 'T' ).AND.
     $         .NOT.LSAME( TRANS, 'C' )       )THEN
        INFO = 1
      ELSE IF( M.LT.0 )THEN
        INFO = 2
      ELSE IF( N.LT.0 )THEN
        INFO = 3
      ELSE IF( LDA.LT.MAX( 1, M ) )THEN
        INFO = 6
      ELSE IF( INCX.EQ.0 )THEN
        INFO = 8
      ELSE IF( INCY.EQ.0 )THEN
        INFO = 11
      END IF
      IF( INFO.NE.0 )THEN
        CALL XERBLA( 'SGEMV ', INFO )
        RETURN
      END IF
*
*     Quick return if possible.
*
      IF( ( M.EQ.0 ).OR.( N.EQ.0 ).OR.
     $    ( ( ALPHA.EQ.ZERO ).AND.( BETA.EQ.ONE ) ) )
     $   RETURN
*
*     Set  LENX  and  LENY, the lengths of the vectors x and y, and set
*     up the start points in  X  and  Y.
*
      IF( LSAME( TRANS, 'N' ) )THEN
        LENX = N
        LENY = M
      ELSE
        LENX = M
        LENY = N
      END IF
      IF( INCX.GT.0 )THEN
        KX = 1
      ELSE
        KX = 1 - ( LENX - 1 )*INCX
      END IF
      IF( INCY.GT.0 )THEN
        KY = 1
      ELSE
        KY = 1 - ( LENY - 1 )*INCY
      END IF
*
*     Start the operations. In this version the elements of A are
*     accessed sequentially with one pass through A.
*     First form  y := beta*y.
*
      IF( BETA.NE.ONE )THEN
```

Procedura 5.7 - Routine SGEMV del livello 2 di BLAS - continua.

```
      IF( INCY.EQ.1 )THEN
        IF( BETA.EQ.ZERO )THEN
          DO 10, I = 1, LENY
            Y( I ) = ZERO
   10     CONTINUE
        ELSE
          DO 20, I = 1, LENY
            Y( I ) = BETA*Y( I )
   20     CONTINUE
        END IF
      ELSE
        IY = KY
        IF( BETA.EQ.ZERO )THEN
          DO 30, I = 1, LENY
            Y( IY ) = ZERO
            IY      = IY   + INCY
   30     CONTINUE
        ELSE
          DO 40, I = 1, LENY
            Y( IY ) = BETA*Y( IY )
            IY      = IY          + INCY
   40     CONTINUE
        END IF
      END IF
    END IF
    IF( ALPHA.EQ.ZERO )
   $  RETURN
    IF( LSAME( TRANS, 'N' ) )THEN
*
*     Form  y := alpha*A*x + y.
*
      JX = KX
      IF( INCY.EQ.1 )THEN
        DO 60, J = 1, N
          IF( X( JX ).NE.ZERO )THEN
            TEMP = ALPHA*X( JX )
            DO 50, I = 1, M
              Y( I ) = Y( I ) + TEMP*A( I, J )
   50       CONTINUE
          END IF
          JX = JX + INCX
   60   CONTINUE
      ELSE
        DO 80, J = 1, N
          IF( X( JX ).NE.ZERO )THEN
            TEMP = ALPHA*X( JX )
            IY   = KY
            DO 70, I = 1, M
              Y( IY ) = Y( IY ) + TEMP*A( I, J )
              IY      = IY      + INCY
   70       CONTINUE
          END IF
          JX = JX + INCX
   80   CONTINUE
      END IF
    ELSE
*
*     Form  y := alpha*A'*x + y.
*
      JY = KY
      IF( INCX.EQ.1 )THEN
        DO 100, J = 1, N
          TEMP = ZERO
```

Procedura 5.7 - Routine SGEMV del livello 2 di BLAS - continua.

```
            DO 90, I = 1, M
              TEMP = TEMP + A( I, J )*X( I )
   90       CONTINUE
            Y( JY ) = Y( JY ) + ALPHA*TEMP
            JY      = JY      + INCY
  100     CONTINUE
        ELSE
          DO 120, J = 1, N
            TEMP = ZERO
            IX   = KX
            DO 110, I = 1, M
              TEMP = TEMP + A( I, J )*X( IX )
              IX   = IX   + INCX
  110       CONTINUE
            Y( JY ) = Y( JY ) + ALPHA*TEMP
            JY      = JY      + INCY
  120     CONTINUE
        END IF
      END IF
*
      RETURN
*
*     End of SGEMV .
*
      END
```

Procedura 5.7 - Routine SGEMV del livello 2 di BLAS - fine.

Dunque la risoluzione di $Ax = b$ può essere eseguita mediante un'operazione *gaxpy* rendendo cosí piú efficiente l'utilizzo dei registri. Bisogna però osservare che la capacità fisica dei registri è limitata. Se N, lunghezza di y, è maggiore di r, capacità dei registri, nell'esecuzione di

$$y \leftarrow y + Mx$$

vengono caricati nei registri blocchi del vettore y, lunghi r, uno alla volta, ottenendo in pratica, una sequenza di istruzioni *saxpy*. Il valore di μ che si ottiene in tal caso è quindi:

$$\mu = \frac{3}{2}$$

caratteristico della routine *saxpy*, cioè del livello 1 di BLAS.

È opportuno in questo caso riorganizzare l'operazione *a blocchi* sul

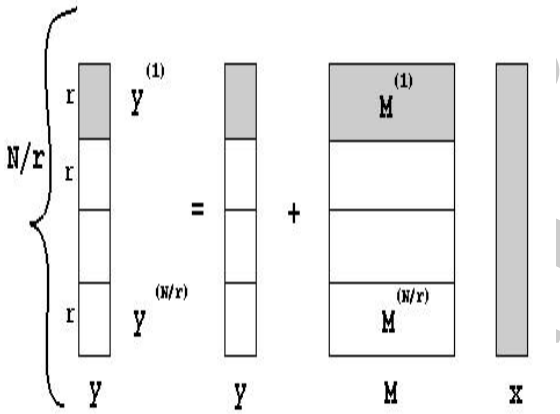


Figura 5.4: *Esecuzione a blocchi dell'operazione* $y \leftarrow y + Mx$. *Nel registro, di capacità* r, *vengono caricate* N/r *righe della matrice* M *e tutto il vettore* x. *Ad ogni passo vengono calcolate* N/r *componenti del vettore* y.

vettore y in modo da forzare il completamento definitivo del calcolo di blocchi di y di lunghezza r, come mostrato nella *Procedura* 5.8.

for $i = 1$ **to** N/r **do**
$y^{(1)} = y^{(1)} + M^{(1)}x$ gaxpy (BLAS2)
$y^{(2)} = y^{(2)} + M^{(2)}x$ gaxpy (BLAS2)
$\vdots$
endfor

Procedura 5.8 - Schema dell'algoritmo a blocchi della gaxpy.

In questo caso, per ogni fissato i si effettuano le operazioni seguenti:

1. si preleva un blocco di y di lunghezza r; l'operazione richiede r accessi;
2. si preleva un blocco di M di dimensione $n \times r$; l'operazione richiede nr accessi;
3. si preleva il vettore x di lunghezza n; l'operazione richiede n accessi;

4. si esegue $y^{(j)} \leftarrow y^{(j)} + M^{(j)}x$; l'operazione richiede $2nr$ operazioni floating point;

5. si ripone il blocco di y di lunghezza r; l'operazione richiede r accessi.

Quindi:

$$\widetilde{\mu}_{gaxpy} = \frac{2r + nr + n}{2nr} = \frac{1}{2} + \frac{1}{2r} + \frac{1}{n}$$

Nel caso migliore, in cui $r = n$, si ha

$$\widetilde{\mu}_{gaxpy} = \frac{1}{2} + \frac{3}{2n} = \mu_{gaxpy}$$

Con calcolatori dotati di un livello di memoria cache si può pensare di riutilizzare i blocchi di matrici presenti nella cache in maniera analoga a come si riutilizzano i vettori nel caso di BLAS2.

Su questo principio si basa il livello 3 di BLAS, dedicato ad operazioni matrice-matrice con la finalità di ridurre ulteriormente il valore di μ.

I principali moduli di BLAS3 sono:

- $C \leftarrow \beta C + \alpha AB$, operazione eseguita dalla routine *sgemm(transa,transb,n,n,n,alpha,a,lda,b,ldb,beta,c,ldc)*
- $B \leftarrow \alpha TB$ (T matrice triangolare), operazione eseguita dalla routine *strmv(side,uplo,trans,diag,n,n,alpha,a,lda,b,ldb)*

```
      SUBROUTINE SGEMM ( TRANSA, TRANSB, M, N, K, ALPHA, A, LDA, B, LDB,
     $                   BETA, C, LDC )
*     .. Scalar Arguments ..
      CHARACTER*1        TRANSA, TRANSB
      INTEGER            M, N, K, LDA, LDB, LDC
      REAL               ALPHA, BETA
*     .. Array Arguments ..
      REAL               A( LDA, * ), B( LDB, * ), C( LDC, * )
*     ..
*  Purpose
*  =======
*
*  SGEMM  performs one of the matrix-matrix operations
*
*     C := alpha*op( A )*op( B ) + beta*C,
*
*  where  op( X ) is one of
*
*     op( X ) = X   or   op( X ) = X',
*
*  alpha and beta are scalars, and A, B and C are matrices, with op( A )
*  an m by k matrix,  op( B )  a  k by n matrix and  C an m by n matrix.
*
*  Parameters
*  ==========
*
*  TRANSA - CHARACTER*1.
*           On entry, TRANSA specifies the form of op( A ) to be used in
*           the matrix multiplication as follows:
*
*              TRANSA = 'N' or 'n',  op( A ) = A.
*
*              TRANSA = 'T' or 't',  op( A ) = A'.
*
*              TRANSA = 'C' or 'c',  op( A ) = A'.
*
*           Unchanged on exit.
*
*  TRANSB - CHARACTER*1.
*           On entry, TRANSB specifies the form of op( B ) to be used in
*           the matrix multiplication as follows:
*
*              TRANSB = 'N' or 'n',  op( B ) = B.
*
*              TRANSB = 'T' or 't',  op( B ) = B'.
*
*              TRANSB = 'C' or 'c',  op( B ) = B'.
*
*           Unchanged on exit.
*
*  M      - INTEGER.
*           On entry,  M  specifies  the number  of rows  of the  matrix
*           op( A )  and of the  matrix  C.  M  must  be at least  zero.
*           Unchanged on exit.
*
*  N      - INTEGER.
*           On entry,  N  specifies the number  of columns of the matrix
*           op( B ) and the number of columns of the matrix C. N must be
*           at least zero.
*           Unchanged on exit.
*
*  K      - INTEGER.
*           On entry,  K  specifies  the number of columns of the matrix
*           op( A ) and the number of rows of the matrix op( B ). K must
```

Procedura 5.9 - Subroutine SGEMM del livello 3 di BLAS - continua

```
*           be at least  zero.
*           Unchanged on exit.
*
*  ALPHA  - REAL            .
*           On entry, ALPHA specifies the scalar alpha.
*           Unchanged on exit.
*
*  A      - REAL             array of DIMENSION ( LDA, ka ), where ka is
*           k  when  TRANSA = 'N' or 'n',  and is  m  otherwise.
*           Before entry with  TRANSA = 'N' or 'n',  the leading  m by k
*           part of the array  A  must contain the matrix  A,  otherwise
*           the leading  k by m  part of the array  A  must contain  the
*           matrix A.
*           Unchanged on exit.
*
*  LDA    - INTEGER.
*           On entry, LDA specifies the first dimension of A as declared
*           in the calling (sub) program. When  TRANSA = 'N' or 'n' then
*           LDA must be at least  max( 1, m ), otherwise  LDA must be at
*           least  max( 1, k ).
*           Unchanged on exit.
*
*  B      - REAL             array of DIMENSION ( LDB, kb ), where kb is
*           n  when  TRANSB = 'N' or 'n',  and is  k  otherwise.
*           Before entry with  TRANSB = 'N' or 'n',  the leading  k by n
*           part of the array  B  must contain the matrix  B,  otherwise
*           the leading  n by k  part of the array  B  must contain  the
*           matrix B.
*           Unchanged on exit.
*
*  LDB    - INTEGER.
*           On entry, LDB specifies the first dimension of B as declared
*           in the calling (sub) program. When  TRANSB = 'N' or 'n' then
*           LDB must be at least  max( 1, k ), otherwise  LDB must be at
*           least  max( 1, n ).
*           Unchanged on exit.
*
*  BETA   - REAL            .
*           On entry,  BETA  specifies the scalar  beta.  When  BETA  is
*           supplied as zero then C need not be set on input.
*           Unchanged on exit.
*
*  C      - REAL             array of DIMENSION ( LDC, n ).
*           Before entry, the leading  m by n  part of the array  C must
*           contain the matrix  C,  except when  beta  is zero, in which
*           case C need not be set on entry.
*           On exit, the array  C  is overwritten by the  m by n  matrix
*           ( alpha*op( A )*op( B ) + beta*C ).
*
*  LDC    - INTEGER.
*           On entry, LDC specifies the first dimension of C as declared
*           in  the  calling  (sub)  program.   LDC  must  be  at  least
*           max( 1, m ).
*           Unchanged on exit.
*
*  Level 3 Blas routine.
*
*  -- Written on 8-February-1989.
*     Jack Dongarra, Argonne National Laboratory.
*     Iain Duff, AERE Harwell.
*     Jeremy Du Croz, Numerical Algorithms Group Ltd.
*     Sven Hammarling, Numerical Algorithms Group Ltd.
*
```

Procedura 5.9 - Subroutine SGEMM del livello 3 di BLAS - continua

```
*     .. External Functions ..
      LOGICAL            LSAME
      EXTERNAL           LSAME
*     .. External Subroutines ..
      EXTERNAL           XERBLA
*     .. Intrinsic Functions ..
      INTRINSIC          MAX
*     .. Local Scalars ..
      LOGICAL            NOTA, NOTB
      INTEGER            I, INFO, J, L, NCOLA, NROWA, NROWB
      REAL               TEMP
*     .. Parameters ..
      REAL               ONE         , ZERO
      PARAMETER          ( ONE = 1.0E+0, ZERO = 0.0E+0 )
*     ..
*     .. Executable Statements ..
*
*     Set  NOTA  and  NOTB  as  true if  A  and  B  respectively are not
*     transposed and set  NROWA, NCOLA and  NROWB  as the number of rows
*     and  columns of  A  and the  number of  rows  of  B  respectively.
*
      NOTA  = LSAME( TRANSA, 'N' )
      NOTB  = LSAME( TRANSB, 'N' )
      IF( NOTA )THEN
        NROWA = M
        NCOLA = K
      ELSE
        NROWA = K
        NCOLA = M
      END IF
      IF( NOTB )THEN
        NROWB = K
      ELSE
        NROWB = N
      END IF
*     Test the input parameters.
*
      INFO = 0
      IF(      ( .NOT.NOTA                 ).AND.
     $         ( .NOT.LSAME( TRANSA, 'C' ) ).AND.
     $         ( .NOT.LSAME( TRANSA, 'T' ) )      )THEN
        INFO = 1
      ELSE IF( ( .NOT.NOTB                 ).AND.
     $         ( .NOT.LSAME( TRANSB, 'C' ) ).AND.
     $         ( .NOT.LSAME( TRANSB, 'T' ) )      )THEN
        INFO = 2
      ELSE IF( M  .LT.0                   )THEN
        INFO = 3
      ELSE IF( N  .LT.0                   )THEN
        INFO = 4
      ELSE IF( K  .LT.0                   )THEN
        INFO = 5
      ELSE IF( LDA.LT.MAX( 1, NROWA ) )THEN
        INFO = 8
      ELSE IF( LDB.LT.MAX( 1, NROWB ) )THEN
        INFO = 10
      ELSE IF( LDC.LT.MAX( 1, M     ) )THEN
        INFO = 13
      END IF
      IF( INFO.NE.0 )THEN
        CALL XERBLA( 'SGEMM ', INFO )
        RETURN
      END IF
*
```

Procedura 5.9 - Subroutine SGEMM del livello 3 di BLAS - continua.

```
*     Quick return if possible.
*
      IF( ( M.EQ.0 ).OR.( N.EQ.0 ).OR.
     $    ( ( ( ALPHA.EQ.ZERO ).OR.( K.EQ.0 ) ).AND.( BETA.EQ.ONE ) ) )
     $    RETURN
*
*     And if  alpha.eq.zero.
*
      IF( ALPHA.EQ.ZERO )THEN
        IF( BETA.EQ.ZERO )THEN
          DO 20, J = 1, N
            DO 10, I = 1, M
              C( I, J ) = ZERO
   10       CONTINUE
   20     CONTINUE
        ELSE
          DO 40, J = 1, N
            DO 30, I = 1, M
              C( I, J ) = BETA*C( I, J )
   30       CONTINUE
   40     CONTINUE
        END IF
        RETURN
      END IF
*
*     Start the operations.
*
      IF( NOTB )THEN
        IF( NOTA )THEN
*
*         Form  C := alpha*A*B + beta*C.
*
          DO 90, J = 1, N
            IF( BETA.EQ.ZERO )THEN
              DO 50, I = 1, M
                C( I, J ) = ZERO
   50         CONTINUE
            ELSE IF( BETA.NE.ONE )THEN
              DO 60, I = 1, M
                C( I, J ) = BETA*C( I, J )
   60         CONTINUE
            END IF
            DO 80, L = 1, K
              IF( B( L, J ).NE.ZERO )THEN
                TEMP = ALPHA*B( L, J )
                DO 70, I = 1, M
                  C( I, J ) = C( I, J ) + TEMP*A( I, L )
   70           CONTINUE
              END IF
   80       CONTINUE
   90     CONTINUE
        ELSE
*
*         Form  C := alpha*A'*B + beta*C
*
          DO 120, J = 1, N
            DO 110, I = 1, M
              TEMP = ZERO
              DO 100, L = 1, K
                TEMP = TEMP + A( L, I )*B( L, J )
  100         CONTINUE
              IF( BETA.EQ.ZERO )THEN
                C( I, J ) = ALPHA*TEMP
              ELSE
```

Procedura 5.9 - Subroutine SGEMM del livello 3 di BLAS - continua.

```
                  C( I, J ) = ALPHA*TEMP + BETA*C( I, J )
                END IF
  110         CONTINUE
  120       CONTINUE
        END IF
      ELSE
        IF( NOTA )THEN
*
*         Form  C := alpha*A*B' + beta*C
*
          DO 170, J = 1, N
            IF( BETA.EQ.ZERO )THEN
              DO 130, I = 1, M
                C( I, J ) = ZERO
  130           CONTINUE
            ELSE IF( BETA.NE.ONE )THEN
              DO 140, I = 1, M
                C( I, J ) = BETA*C( I, J )
  140           CONTINUE
            END IF
            DO 160, L = 1, K
              IF( B( J, L ).NE.ZERO )THEN
                TEMP = ALPHA*B( J, L )
                DO 150, I = 1, M
                  C( I, J ) = C( I, J ) + TEMP*A( I, L )
  150             CONTINUE
              END IF
  160         CONTINUE
  170       CONTINUE
        ELSE
*
*         Form  C := alpha*A'*B' + beta*C
*
          DO 200, J = 1, N
            DO 190, I = 1, M
              TEMP = ZERO
              DO 180, L = 1, K
                TEMP = TEMP + A( L, I )*B( J, L )
  180           CONTINUE
              IF( BETA.EQ.ZERO )THEN
                C( I, J ) = ALPHA*TEMP
              ELSE
                C( I, J ) = ALPHA*TEMP + BETA*C( I, J )
              END IF
  190         CONTINUE
  200       CONTINUE
        END IF
      END IF
*
      RETURN
*
*     End of SGEMM .
*
      END
```

Procedura 5.9 - Subroutine SGEMM del livello 3 di BLAS - fine.

Utilizzando *sgemm* vengono effettuate le operazioni seguenti:

1. si preleva C; l'operazione richiede n^2 accessi;
2. si preleva A; l'operazione richiede n^2 accessi;
3. si preleva B; l'operazione richiede n^2 accessi
4. si esegue $C \leftarrow C + AB$; l'operazione richiede $2n^3$ operazioni floating point;
5. si ripone C; l'operazione richiede n^2 accessi.

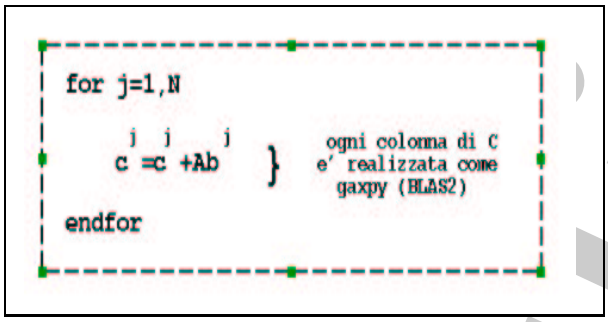
In definitiva, risulta:

$$\mu_{sgemm} = \frac{4n^2}{2n^3} = \frac{2}{n} < \frac{3}{2n} + \frac{1}{2} \approx \mu_{gaxpy}$$

Si è quindi avuta un'ulteriore riduzione di μ.

Analogamente a quanto accade per BLAS2, il problema maggiore per un utilizzo efficace delle routine di BLAS3 è la capacità limitata della cache, per cui se l'area di memoria richiesta dai dati (in particolare della matrice) è piú grande della capacità della memoria cache, questi verranno necessariamente suddivisi e quindi prelevati dalla memoria piú volte. In pratica è come avere una sequenza di *gaxpy* sulle colonne della matrice C come descritto nella *Procedura* 5.10 per la quale risulta

$$\mu_{sgemm} \approx \mu_{gaxpy}$$



Procedura 5.10 - Algoritmo per il calcolo del prodotto matrice per matrice utilizzando la routine gaxpy di BLAS2.

Una soluzione a tale problema può essere quella di riformulare l'algoritmo organizzando esplicitamente le operazioni a *"blocchi"*. In particolare, si suddividono le matrici in p blocchi di righe e di colonne, ognuno di dimensione $\omega \times \omega$ e si effettuano gli aggiornamenti su blocchi di C (*Procedura* 5.11).

```
for ... = 1 to p do
    for ... = 1 to p do
        for ... = 1 to p do
            C_ij = C_ij + A_ik B_kj
        endfor
    endfor
endfor
```

Procedura 5.11 - Schema dell'algoritmo a blocchi per il calcolo del prodotto matrice per matrice.

Il nucleo computazionale è ancora un prodotto matrice per matrice. In questo caso vengono eseguite le operazioni seguenti:

1. si preleva C_{ij} di dimensione $\omega \times \omega$; l'operazione richiede ω^2 accessi;
2. si preleva $A_{ik} \in \Re^{\omega \times \omega}$ $(k = 1, ..., p)$; l'operazione richiede $p\omega^2$ accessi;

3. si preleva $B_{kj} \in \Re^{\omega \times \omega}$ $(k = 1, ..., p)$; l'operazione richiede $p\omega^2$ accessi;

4. si esegue $C_{ij} = C_{ij} + A_{ik}B_{kj}$; l'operazione richiede $2p\omega^3$ operazioni floating point;

5. si ripone $C_{ij} \in \Re^{\omega \times \omega}$; l'operazione richiede ω^2 accessi.

Il parametro μ per la subroutine sgemm a blocchi è il seguente:

$$\widetilde{\mu}_{sgemm} = \frac{2\omega^2 + 2p\omega^2}{2p\omega^3} = \frac{p+1}{p\omega} = \frac{1}{n} + \frac{1}{\omega}$$

Osserviamo che nel caso *"ideale"* in cui $\omega = n$ allora

$$\widetilde{\mu}_{sgemm} = \frac{2}{n}$$

Riassumendo, nella Tab. 5.1 sono riportate le caratteristiche dei tre livelli di BLAS.

	BLAS1	*BLAS2*	*BLAS3*
tipo operazione	$y \leftarrow y + \alpha x$	$y \leftarrow y + Mx$	$C \leftarrow C + AB$
#accessi in memoria	$O(n)$	$O(n^2)$	$O(n^3)$
#operazioni fl.p.	$3n + 1$	$3n + n^2$	$4n^2$
μ	$3/2$	$1/2$	$2/n$

Tabella 5.1: Nella tabella sono sintetizzate le caratteristiche principali dei tre livelli di BLAS

Le performance dei tre livelli sono riportate in Fig. 5.5. Il grafico riporta le performance di BLAS su una macchina RISC 6000/590 in Megaflops al variare della dimensione della matrice o del vettore. La curva superiore è per il prodotto matrice per matrice (BLAS3); la curva centrale è per il prodotto matrice per vettore (BLAS2) e la curva piú bassa è la *saxpy* (BLAS1). Come si vede, si ha un forte

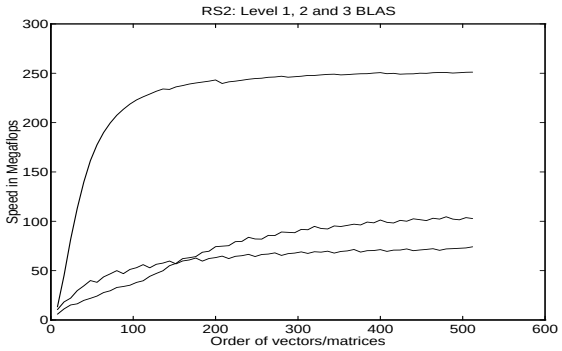


Figura 5.5: *Performance dei tre livelli di BLAS su un processore RISC6000/590.*

incremento di velocità se si scrive un algoritmo in termini di BLAS3. Riferita al LINPACK benchmark, la *peak performance* di BLAS3, circa 250 Mflops, è vicina alla *peak performance* della macchina cioè circa 266 Mflops.

5.2 Da LINPACK a ScaLAPACK

Le routine di BLAS sono alla base della maggior parte del software per il calcolo matriciale numerico. Uno dei primi package di software matematico è EISPACK (1972-1973), una raccolta di software scritto in Fortran per il calcolo di autovalori e/o autovettori di matrici generali complesse o reali, Hermitiane complesse, simmetriche reali, simmetriche a bande reali, simmetriche tridiagonali reali, tridiagonali speciali reali, generalizzate reali ed in fine reali simmetriche generalizzate.

Nel 1979 nasce LinPack (**Lin**ear **Pack**age), una libreria di software per il calcolo matriciale i cui nuclei computazionali sono basati su *building-blocks* del livello 1 di BLAS. Il package risolve sistemi lineari le cui matrici possono essere piene, a banda, simmetriche indefini-

te, simmetriche definite positive, triangolari e quadrate tridiagonali. Tali routine calcolano la fattorizzazione di una matrice secondo gli schemi: LU, di Cholesky con pivoting, QR e SVD. Le routine di LinPack permettono anche di calcolare l'inversa, il determinante e l'indice di condizionamento di una matrice.

L'evoluzione delle architetture a memoria gerarchica ha reso EISPACK e LinPack inefficienti. È nata cosí la necessità di riformulare EISPACK e LinPack in termini di BLAS2 e BLAS3.

Nel 1987 nasce LAPack (**L**inear **A**lgebra **Pack**age), il package che utilizza algoritmi a blocchi per risolvere problemi quali:

- Risoluzione di sistemi lineari,
- Problemi di minimi quadrati,
- Problemi agli autovalori,
- Problemi ai valori singolari.

Si possono fornire in input matrici generiche dense, a banda, simmetriche, Hermitiane (definite positive) o triangolari. I *building-blocks* di LAPACK sono routine di BLAS2 e BLAS3.

Successivamente, con l'avvento dei calcolatori paralleli è nata l'esigenza di realizzare una libreria che estendesse le potenzialità di LAPACK a calcolatori MIMD a memoria distribuita. Nasce cosí il progetto ScaLAPACK (**Sca**lable **LAPACK**) che comprende sia una versione parallela dei tre livelli di BLAS, PBLAS (**P**arallel **BLAS**), che una libreria di message-passing specializzata per matrici e vettori, BLACS (**B**asic **L**inear **A**lgebra **C**omunication **S**ubprograms).

5.2.1 BLACS

BLACS opera su due classi di matrici. La prima classe è costituita dalle matrici rettangolari, memorizzate come array bidimensionali composti da M righe ed N colonne, e di leading dimension LDA,

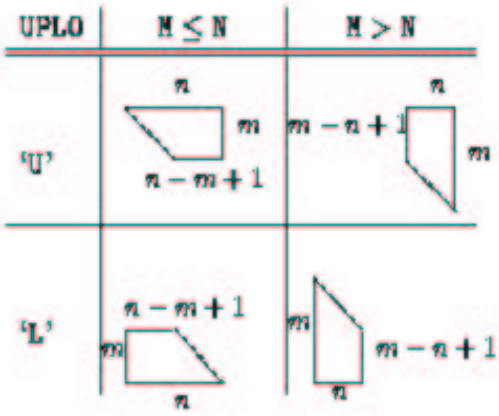


Figura 5.6: *In figura sono rappresentate le varie matrici trapezoidali che si ottengono al variare dei parametri M, N ed $UPLO$.*

che determina la distanza in memoria tra due successivi elementi di una riga (VBLACS assume che gli array siamo memorizzati per colonne).

La seconda classe di matrici è costituita dalle matrici trapezoidali. Come per la classe delle matrici rettangolari, le matrici trapezoidali sono definite da M, N e LDA. A questi si aggiunge il parametro $UPLO$ che indica se la matrice è trapezoidale superiore o inferiore, e $DIAG$ che specifica se la diagonale della matrice è unitaria. Le matrici triangolari sono una sottoclasse delle matrici trapezoidali. In Fig. 5.6 sono riportati i vari tipi di matrici trapezoidali al variare dei parametri M, N ed $UPLO$.

Gli N_p processi coinvolti in una operazione concorrente sono in genere presentati come un array monodimensionale di processi, ciscuno identificato da un intero $N = 0, 1, 2, ..., N_p - 1$. Talvolta può essere opportuno trasformare l'array di processi ad una dimensione in una griglia bidimensionale, in tal caso, la griglia può essere costituita da P righe e Q colonne dei processi, con $P * Q = N_g \leq N_p$. In questo modo si può individuare un processo, oltre che con l'i-

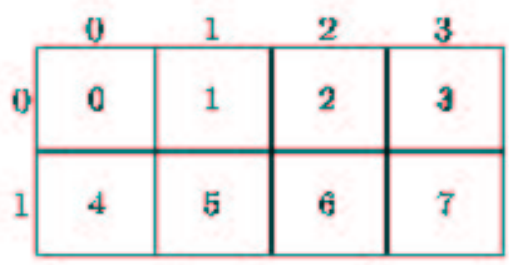


Figura 5.7: 8 *processi organizzati in una griglia di processi di dimensione* 2×4.

dentificativo N, anche utilizzando le sue coordinate nella griglia $(p, q), con\ \ 0 \leq p < P, e\ \ 0 \leq q < Q$. Un esempio di *Griglia* dei processi è mostrato in Fig. 5.7.

Questa organizzazione logica dei processi è molto utile nelle *scoped operations*, cioè nelle operazioni in cui sono coinvolti piú di un mittente e piú di un destinatario in un *contesto (scoped)* della griglia. In un sistema che utilizza un array monodimensionale di processi, l'unico contesto possibile è quello che comprende tutti i processi. In una griglia bidimensionale di processi i contesti possibili sono tre : *Riga, Colonna e Tutti*, come mostrato nella Tab. 5.2.

CONTESTO	SIGNIFICATO
Riga	Partecipano tutti i processi appartenenti ad una riga
Colonna	Partecipano tutti i processi apparteneti ad una colonna
Tutti	Partecipano tutti i processi

Tabella 5.2: Nelle colonne della tabella sono indicati il nome dei Contesti ed il loro significato.

Le BLACS supportano inoltre comunicazioni uno-a-tutti, le cosiddette *broadcast*, ad *anello* e ad *albero*: le prime sono indicate quando si ha interesse a *"liberare"* quanto prima possibile il processore sorgente della comunicazione, ossia quello che invia dati agli altri coinvolti nella suddetta operazione, oppure quando la rete non è adatta a comunicazioni concorrenti (ad esempio collegamenti Ether-

net); il tempo di attesa complessivo è $(p-1)t_{com}$ con p processori, ed i processori vengono sincronizzati, per cui è indicata quando si devono eseguire due broadcast successivi. Il broadcast ad albero è, d'altra parte, piú veloce, nel senso che viene completato in un tempo $(\log_2(p))\ t_{com}$, ma piú messaggi viaggiano contemporaneamente ad ogni passo.

5.2.2 ScaLAPACK

ScaLAPACK, completata alla fine del 1994, estende l'utilizzo della libreria LAPACK a calcolatori MIMD a memoria distribuita, utilizzando chiamate a BLACS per le comunicazioni. Requisiti di questo progetto sono l'efficienza, la scalabilità, affidabilità e la facilità di utilizzo. Molti di questi, specialmente la portabilità, vengono raggiunti sviluppando e utilizzando degli standard per le routine di comunicazione e di calcolo. La Fig. 5.8 descrive la gerarchia del software di ScaLAPACK. Le componenti sotto la linea tratteggiata (local) sono chiamate su di un singolo processore, con argomenti memorizzati solo su un processore. Le componenti che si trovano sopra la linea tratteggiata (global) sono routine parallele sincrone, i cui argomenti includono matrici e vettori distribuiti a blocchi su piú processori.

ScaLAPACK utilizza PBLAS, con un'interfaccia il piú possibile simile a quella di BLAS. Questa scelta ha reso il codice di ScaLAPACK molto simile, a volte identico, all'analogo codice di LAPACK. È stata aggiunta solo una routine per la trasposizione di una matrice, vista la complessità di questa operazione in ambiente a memoria distribuita.

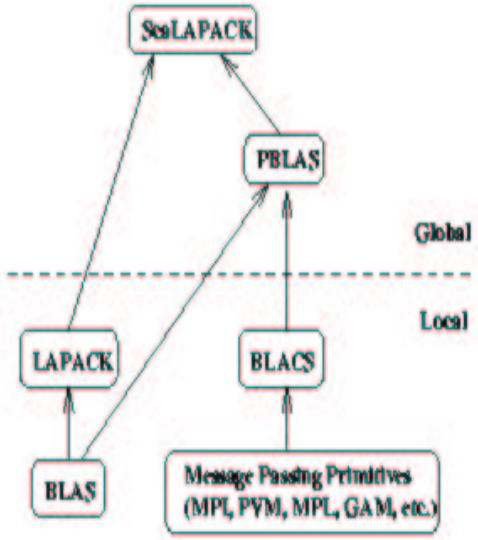


Figura 5.8: *La figura descrive l'organizzazione gerarchica del software di ScaLAPACK. Le componenti sotto la linea tratteggiata (local) sono chiamate dai singoli processori. Le componenti sopra la linea tratteggiata (global) sono routine parallele, i cui argomenti comprendono matrici e vettori distribuiti per blocchi in modo periodico su processori organizzati in una griglia bidimensionale.*

Di seguito è riportata la chiamata alla routine di BLAS che effettua il prodotto matrice-matrice in doppia precisione, DGEMM, e la rispettiva routine di PBLAS, PDGEMM. È da notare quanto siano simili.

```
CALL DGEMM ( TRANSA, TRANSB, M, N, ALPHA,
                                    A( IA, JA), LDA,
                                    B( IB, JB), LDB, BETA,
                                    C( IC, JC), LDC)
CALL PDGEMM ( TRANSA, TRANSB, M, N, ALPHA,
                                    A, IA, JA, DESC_A,
                                    B, IB, JB, DESC_B, BETA,
                                    C, IC, JC, DESC_C)
```

Per passare a DGEMM la sottomatrice che inizia da A(IA, JA), ad esempio, l'argomento attuale corrispondente all'argomento formale A, è A(IA, JA). In PDGEMM, invece, per poter estrarre una sottomatrice da A si devono fornire dati che definiscono lo schema globale di memorizzazione, quindi IA e JA devono essere passati separatamente. DESC_A è l'array descrittore della matrice A. I parametri che descrivono le matrici B e C sono analoghi a quelli di A.

5.3 La fattorizzazione LU su architetture MIMD a memoria distribuita

5.3.1 La versione Block-partitioned di LAPACK

Riprendiamo l'algoritmo di eliminazione di Gauss. Abbiamo visto che permutando l'ordine dei tre cicli *for* si hanno 6 varianti, ciascuna caratterizzata da un'operazione di base eseguita piú volte al variare degli indici. Piú precisamente:

$$\left.\begin{array}{l} ijk \\ jik \end{array}\right\} \Longrightarrow prodotto\ scalare \Rightarrow BLAS1$$

$$\left.\begin{array}{l} kij \\ kji \end{array}\right\} \Longrightarrow saxpy \Rightarrow BLAS1$$

$$\left.\begin{array}{l} ikj \\ jki \end{array}\right\} \Longrightarrow gaxpy \Rightarrow BLAS2$$

Tali versioni dell'algoritmo di Gauss sono anche note in letteratura rispettivamente come *Crout, right-looking* e *left-looking*. L'algoritmo block-partitioned implementato in LAPACK, cosí come la versione parallela in SCALAPACK, è basato sulla seconda, cioè la *right-looking*; mentre la *left-looking* garantisce una gestione migliore della memoria, prestandosi, tra l'altro, a implementazioni I/O tolerant.

È noto che il metodo di eliminazione di Gauss realizza la fattorizzazione LU della matrice A

$$A = LU$$

con L matrice triangolare inferiore i cui elementi diagonali sono uguali ad 1 e U matrice triangolare superiore.

Fattorizzazione LU con pivoting parziale

```
⋮
% ciclo sui passi
for k = 1 to n − 1 do
% scelta del pivot al passo k
   a_rk := max(|a_ik|), r > i ≥ k
   if (|a_rk| ≠ 0) then
     for j = k to n do
       scambiare a_kj con a_rj
     endfor
% cicli per ottenere le matrici L ed U
     for i = k + 1 to n do
% calcolo elementi della matrice L
       a_ik := a_ik/a_kk
% calcolo degli elementi della matrice U
       for j = k + 1 to n do
         a_ij = a_ij − a_ik a_kj
       endfor
     endfor
   else
     A singolare
endfor
⋮
```

Procedura 5.12 - Algoritmo della fattorizzazione LU con pivoting parziale.

Nel discutere i tre livelli della libreria BLAS abbiamo visto che il terzo livello è il piú efficiente in quanto tiene conto della struttura gerarchica della memoria. È quindi opportuno utilizzare BLAS3 nell'algoritmo di eliminazione di Gauss, riorganizzandolo in termini di algoritmi a blocchi. L'idea alla base della versione a blocchi della fattorizzazione LU è quella di trasformare la matrice considerando

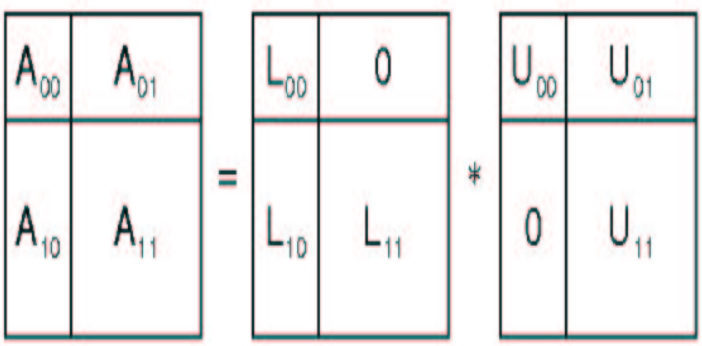


Figura 5.9: *Fattorizzazione a blocchi LU della matrice A. A_{00} è una matrice di dimensione $b \times b$, A_{01} ha dimensione $b \times (n-b)$, A_{10} ha dimensione $(n-b) \times b$ e la matrice A_{11} ha dimensione $(N-b) \times (N-b)$. L_{00} e L_{11} sono matrici triangolari inferiori con elementi diagonali uguali a 1 e U_{00} e U_{11} sono matrici triangolari superiori.*

blocchi di b colonne, invece che una sola colonna alla volta. Il parametro b è detto *block size.* Prima viene effettuata la fattorizzazione delle b colonne del blocco della matrice attiva. Successivamente, viene aggiornata la matrice attiva rimanente. Il valore ottimale di b dipende dalle caratteristiche dell'ambiente di calcolo e deve essere scelto in modo da massimizzare le prestazioni dell'algoritmo. Affinché b risulti ottimale deve avere le caratteristiche seguenti:

- deve essere piccolo abbastanza in modo tale che le b colonne che devono essere fattorizzate entrino tutte nel livello di memoria veloce (ad esempio nella memoria cache);
- deve essere grande abbastanza in modo tale che il prodotto matrice $\times$ matrice sia effettuato in modo efficiente utilizzando ad esempio moduli di BLAS3.

Supponiamo che la matrice $A \in \Re^{n \times n}$ sia partizionata come mostrato nella Fig. 5.9.
Dall'essere

$$A = LU$$

risulta:

$$\begin{aligned} A_{00} &= L_{00}U_{00} \\ A_{10} &= L_{10}U_{00} \\ A_{01} &= L_{00}U_{01} \\ A_{11} &= L_{10}U_{01} + L_{11}U_{11} \end{aligned}$$

dove

$$A_{00} \in \Re^{b\times b} \quad A_{01} \in \Re^{b\times(n-b)} \quad A_{10} \in \Re^{(n-b)\times b} \quad A_{11} \in \Re^{(n-b)\times(n-b)}$$

L_{00} e L_{11} sono matrici triangolari inferiori con elementi diagonali uguali a 1 e U_{00} e U_{11} sono matrici triangolari superiori.

Le prime due equazioni rappresentano la fattorizzazione LU del primo blocco di dimensione $n \times b$ della matrice A. Calcolate le matrici L_{00}, L_{10} e U_{00} è possibile risolvere la terza equazione rispetto a U_{01}, ovvero risolvere il sistema triangolare inferiore multiplo:

$$L_{00}U_{01} = A_{01}$$

le cui incognite sono le $n-b$ colonne della matrice U_{01} e i termini noti le $n-b$ colonne di A_{01}. Infine, dall'ultima equazione segue che:

$$A'_{11} = A_{11} - L_{10}U_{01} = L_{11}U_{11}$$

da cui si ricavano L_{11} e U_{11} effettuando la fattorizzazione LU della matrice $A'_{11} \in \Re^{(n-b)\times(n-b)}$. Questo può essere fatto riapplicando l'algoritmo alla matrice A'_{11}. Posto

$$k = \left\lceil \frac{n}{b} \right\rceil$$

uno schema dei passi dell'algoritmo a blocchi per la fattorizzazione LU della matrice A è descritto nella *Procedura* 5.13 .

Fattorizzazione LU

for $ib = 1$ **to** k **do**
% calcola la fattorizzazione LU di A_{00} e A_{10}
$A_{00} = L_{00}U_{00}$
$A_{10} = L_{10}U_{00}$
% calcolo di U_{01}
$L_{00}U_{01} = A_{01}$
% calcolo di L_{11} e U_{11}
$A'_{11} = L_{11}U_{11}$
endfor

Procedura 5.13 - Schema dell'algoritmo a blocchi che esegue la fattorizzazione LU.

In dettaglio, l'algoritmo per la fattorizzazione LU è illustato nella *Procedura* 5.14 .

Fattorizzazione LU con pivoting parziale - implementazione a blocchi

$\vdots$
for $ib = 1$ **to** $n-1$ **step** b **do**
% indice dell'ultima colonna del blocco attivo
$end = min(ib + b - 1, n)$
% fattorizzazione LU del blocco $A_{00} = A(ib : n, ib : end)$
for $i = ib$ **to** end **do**
% scelta del pivot
$a_{ri} := max(|a_{ji}|), r > j \geq i$
if $(|a_{ri}| \neq 0)$ **then**
for $j = k$ **to** n **do**
scambiare a_{kj} con a_{rj}
endfor
endif
% calcolo degli elementi delle matrici L_{00} e L_{10}
for $j = i + 1$ **to** n **do**
$a_{ji} = a_{ji}/a_{ii}$
% calcolo degli elementi della matrice U_{00}
for $k = i + 1$ **to** end **do**
$a_{jk} = a_{jk} - a_{ji} * a_{ik}$
endfor
endfor
endfor

Procedura 5.14 - Algoritmo per la fattorizzazione LU con pivoting parziale (implementazione a blocchi) - continua

% sia L_{00} la matrice triangolare inferiore di dimensione $b \times b$ i cui elementi
% sotto la diagonale principale sono memorizzati in $A(ib:end, ib:end)$ e i cui
% elementi diagonali sono uguali ad 1. Aggiornamento della matrice
% U_{01} memorizzata in $A(ib:end, end+1:n)$ mediante risoluzione
% di $n - end$ sistemi triangolari per il calcolo
% della matrice U_{01} che sovrascrive $A(ib:end, end+1:n)$
$A(ib:end, end+1:n) \leftarrow solve\,(L_{00} * U_{01} = A(ib:end, end+1:n))$
% aggiornamento della matrice $A(end+1:n, end+1:n)$
$A(end+1:n, end+1:n) = A(end+1:n, end+1:n) +$
$-A(end+1:n, ib:end) * A(ib:end, end+1:n)$
endfor
⋮

Procedura 5.14 - Algoritmo per la fattorizzazione LU con pivoting parziale (implementazione a blocchi) - fine

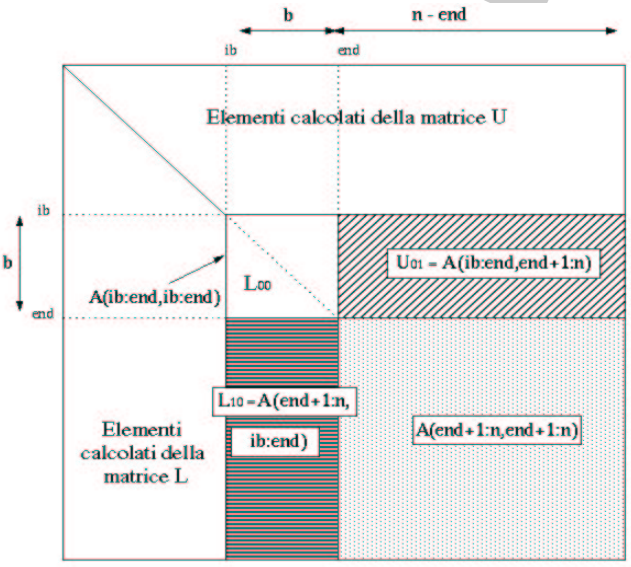


Figura 5.10: *Fattorizzazione LU della matrice A. Il blocco U_{01} è ottenuto risolvendo $n - end$ sistemi triangolari la cui matrici dei coefficienti è la matrice triangolare inferiore L_{00} di dimensione $b \times b$. La sottomatrice attiva è ottenuta sottraendo a se stessa il prodotto del blocco L_{01} per il blocco U_{01}.*

♣ Esempio 27

Sia $A \in \Re^{N \times N}$ e sia $n = N/3$. Partizioniamo la matrice in 3×3 blocchi ciascuno di dimensione $n \times n$; la fattorizzazione LU può esprimersi nel modo seguente:

$$\begin{pmatrix} A_{11} & A_{12} & A_{13} \\ A_{21} & A_{22} & A_{23} \\ A_{31} & A_{32} & A_{33} \end{pmatrix} = \begin{pmatrix} L_{11} & 0 & 0 \\ L_{21} & L_{22} & 0 \\ L_{31} & L_{32} & L_{33} \end{pmatrix} \cdot \begin{pmatrix} U_{11} & U_{12} & U_{13} \\ 0 & U_{22} & U_{23} \\ 0 & 0 & U_{33} \end{pmatrix}$$

con L_{11}, L_{22}, L_{33} triangolari inferiori unitarie, U_{11}, U_{22}, U_{33} triangolari superiori [3].
Effettuando il prodotto e uguagliando blocco a blocco otteniamo

$$\begin{array}{lll} A_{11} = L_{11}U_{11} & A_{12} = L_{11}U_{12} & A_{13} = L_{11}U_{13} \\ A_{21} = L_{21}U_{11} & A_{22} = L_{21}U_{12} + L_{22}U_{22} & A_{23} = L_{21}U_{13} + L_{22}U_{23} \\ A_{31} = L_{31}U_{11} & A_{32} = L_{31}U_{12} + L_{32}U_{22} & A_{33} = L_{31}U_{13} + L_{32}U_{23} + L_{33}U_{33} \end{array}$$

una prima procedura può essere la seguente:

PASSO 1

1. calcola la fattorizzazione LU del pannello

$$\begin{pmatrix} A_{11} \\ A_{21} \\ A_{31} \end{pmatrix} \rightarrow \begin{pmatrix} L_{11}U_{11} \\ L_{21}U_{11} \\ L_{31}U_{11} \end{pmatrix}$$

2. $\left.\begin{array}{l} A_{12} = L_{11}U_{12} \\ A_{13} = L_{11}U_{13} \end{array}\right\} \Rightarrow calcola \begin{array}{l} U_{12} = L_{11}^{-1}A_{12} \\ U_{13} = L_{11}^{-1}A_{13} \end{array}$;

[3] Osserviamo che un algoritmo *block-partitioned*, a differenza di un algoritmo a blocchi, consiste semplicemente in una riorganizzazione dei calcoli, ma di fatto non differisce dal corrispondente algoritmo scalare; ad esempio, la fattorizzazione $\bar{L}\bar{U}$ a blocchi di una matrice A genera 2 fattori $\bar{L}$ ed $\bar{U}$ rispettivamente triangolare inferiore e triangolare superiore a blocchi, mentre le matrici L ed U sono di fatto le medesime calcolate dall'algoritmo che denominiamo "classico" in contrapposizione a quello *block-partitioned*. Questo fa sí che si ereditino le proprietà di stabilità dell'algoritmo scalare: in particolare si può dimostrare [27] che per una generica matrice rettangolare B, $M \times N$, naturalmente nell'ipotesi di applicabilità della fattorizzazione LU, si ha che i fattori calcolati $\tilde{L}, \tilde{U}$ soddisfano:

$$B = \tilde{L}\tilde{U} + H$$

con

$$|H| \leq 3 \cdot (min\{M, N\} - 1) \cdot \epsilon \cdot \left(|B| + \left|\tilde{L}\right| \left|\tilde{U}\right|\right) + O(\epsilon^2)$$

dove con ϵ indichiamo la massima accuratezza relativa de sistema aritmetico del calcolatore. L'errore può pertanto essere controllato attraverso la tecnica del *pivoting* mediante la quale otteniamo che tutti gli elementi della matrice $\tilde{L}$ abbiano modulo≤ 1.

3. aggiorna la matrice attiva:

$$\begin{pmatrix} A_{22} & A_{23} \\ A_{32} & A_{33} \end{pmatrix} \rightarrow \begin{pmatrix} A_{22}^{-} = A_{22} - L_{21}U_{12} & A_{23}^{-} = A_{23} - L_{21}U_{13} \\ A_{32}^{-} = A_{32} - L_{31}U_{12} & A_{33}^{-} = A_{33} - L_{31}U_{13} \end{pmatrix}$$

PASSO 2

1. calcola la fattorizzazione LU del pannello

$$\begin{pmatrix} A_{22}^{-} \\ A_{32}^{-} \end{pmatrix} \rightarrow \begin{pmatrix} L_{22}U_{22} \\ L_{32}U_{22} \end{pmatrix}$$

2. calcola $A_{23}^{-} = L_{22}U_{23} \;\Rightarrow\; U_{23} = L_{22}^{-1}A_{23}^{-}$;
3. aggiorna la matrice attiva:

$$\tilde{A}_{33} = A_{33}^{-} - L_{32}U_{23}$$

PASSO 3
calcola la fattorizzazione LU di $\tilde{A}_{33} = L_{33}U_{33}$.

△

Nella *Procedura* 5.15 è schematizzato l'algoritmo della fattorizzazione LU a blocchi per una matrice A quadrata, di dimensione n e partizionata in blocchi $[b \times b]$.

```
for k = 1 to n step b do
    applica la fatt. LU al pannello A(k : n, k : k + b − 1)
    calcola il pannello U(k : k + b − 1, k + b : n)
    aggiorna la matrice attiva
endfor
```

Procedura 5.15 - Schema dell'algoritmo della fattorizzazione LU a blocchi $[b \times b]$ di una matrice quadrata.

Osserviamo che se A è una matrice rettangolare di dimensioni $[m \times n]$ la generalizzazione dello schema descritto si ottiene facilmente (*Procedura* 5.16).

```
% calcola il minimo tra il numero di righe e di colonne della matrice A
s =min{m, n}
% effettua la fattorizzazione LU della matrice A
% in un numero s/b di passi
for k = 1 to s step b do
% calcola il minimo tra il numero di colonne da aggiornare
% e la dimensione di blocco
    kb = min{s − k + 1, b}
    applica la fatt. LU al pannello A(k : m, k : k + kb − 1)
    calcola il pannello U(k : k + kb − 1, k + kb : n)
    aggiorna la matrice attiva A(k + kb : m; k + kb : n)
endfor
```

Procedura 5.16 - Schema dell'algoritmo della fattorizzazione LU a blocchi $[kb \times kb]$ di una matrice rettangolare.

Nella *Procedura* 5.17 viene integrato l'algoritmo right-looking con il pivoting parziale.

```
% calcola il minimo tra il numero di righe e di colonne della matrice A
s =min{m, n}
for k = 1 to s step b do
% calcola il minimo tra il numero di colonne da aggiornate
% e la dimensione di blocco
    kb = min{s − k + 1, b}
    applica la fatt. LU con pivoting parziale al pannello A(k : m, k : k + kb − 1)
    applica gli scambi alle righe del pannello A(k : m; 1 : k − 1)
    che precede quello attivo al passo corrente
    applica gli scambi alle righe del pannello A(k : m; k + kb : n)
    che segue quello attivo al passo corrente
    calcola il pannello U(k : k + kb − 1, k + kb : n)
    aggiorna la matrice attiva A(k + kb : m; k + kb : n)
endfor
```

Procedura 5.17 - Schema dell'algoritmo della fattorizzazione LU a blocchi $[kb \times kb]$ con pivoting parziale di una matrice rettangolare.

Nell'algoritmo di fattorizzazione LU right-looking individuiamo, in corrispondenza di ogni fase di ciascun passo, i nuclei computazionali fondamentali seguenti:

1. fattorizzazione LU di un pannello rettangolare;
2. calcolo di un blocco di righe di U: risoluzione di sistemi lineari triangolari inferiori a termine noto multiplo;
3. aggiornamento della matrice attiva: prodotto matrice per matrice.

Nella routine di LAPACK preposta al calcolo della fattorizzazione LU di una matrice le tre fasi sono espletate attraverso l'uso di tre moduli di BLAS; in particolare, quella relativa alla fase 1. fa uso di moduli di BLAS2, le altre due richiamano invece moduli di BLAS3.

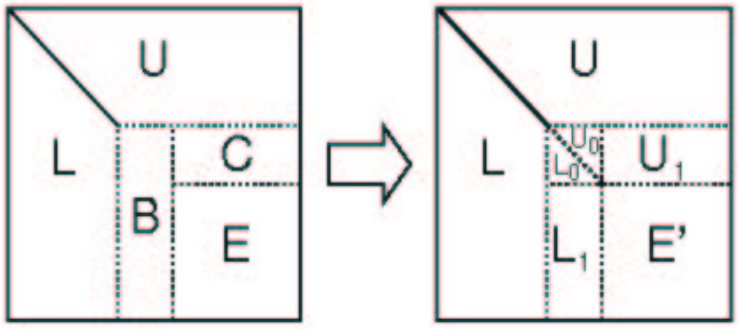


Figura 5.11: *La figura mostra come al passo* $k+1$ *dell'algoritmo di fattorizzazione LU a blocchi, i pannelli B, C e E vengono aggioranati. Le sottomatrici trapezoidali LU e U sono state già valutate nei passi precedenti. L è composta da kb colonne e U è composta da kb righe.*

Vediamo ora l'algoritmo in termini di chiamate a BLAS2 e BLAS3. Nell'algoritmo si fa riferimento ai blocchi della matrice A come rappresentati nella Fig. 5.11.

for $k = 1$ **to** n/b **do**
fattorizza il pannello B con pivoting
% per il calcolo dell'indice del massimo elemento
% nella colonna corrente si utilizza la funzione:
% FUNCTION IxAMAX(N, X, INCX)
% per lo scambio di sue righe del pannello B si
% utilizza la routine:
% xSWAP(N,X,INCX,Y,INCY)
% per l'aggiornamento della matrice attiva
% per l'esecuzione della gaxpy si utilizza:
% xGemv(TRANS,M,N,ALPHA,A,LDA,X,INCX,BETA,Y,INCY)
risolve $U_1 = L_1^{-1}C$
% per la risoluzione del sistema lineare, con matrice

Procedura 5.18 - Schema dell'algoritmo della fattorizzazione LU con pivoting parziale in termini di chiamate alla libreria BLAS - continua

```
    % triangolare inferiore utilizza la routine:
    % xTRSM(SIDE,UPLO,TRANSA,DIAG,M,N,ALPHA,A,LDA,B,LDB)

    aggiorna E' = E - L1 * U1
    % per l'aggiornamento della matrice attiva utilizza la routine:
    % xGEMM(TRANSA,TRANSB,M,N,K,ALPHA,A,LDA,B,LDB,BETA,C,LDC)

endfor
```

Procedura 5.18 - Schema dell'algoritmo della fattorizzazione LU con pivoting parziale in termini di chiamate alla libreria BLAS - fine

5.3.2 Distribuzione ciclica a blocchi nella Fattorizzazione LU

Nel Cap. 3, nel descrivere gli algoritmi per il calcolo del prodotto matrice per vettore, è stata introdotto la *distribuzione ciclica* della matrice lungo le righe o lungo le colonne. In particolare, nell'esempio 9 si utilizza una *Distribuzione Ciclica a Blocchi di Righe* in cui le righe della matrice $A \in \Re^{n \times n}$ sono suddivise in blocchi di dimensione nb e poi, ciascun blocco viene assegnato ad un processore, in modo che la k-esima riga di A appartenga al processore il cui identificativo è

$$id = mod(\lfloor k/nb \rfloor, p)$$

dove p indica il numero di processori ed il simbolo $\lfloor x \rfloor$ la parte intera di x. Nell'esempio 9 $nb = 1$ e $p = 2$.
Nell'esempio 10 si utilizza una *Distribuzione Ciclica a Blocchi di Colonne* in cui le colonne della matrice $A \in \Re^{n \times n}$ sono prima divise in blocchi di dimensione nb e poi, ciascun blocco, viene assegnato ad un processore, in modo che la k-esima colonna di A appartenga

al processore il cui identificativo è

$$id = mod(\lfloor k/nb \rfloor, p)$$

dove il simbolo $\lfloor x \rfloor$ la parte intera di x. Nell'esempio 10 $nb = 1$ e $p = 2$.

Introduciamo ora la distribuzione ciclica a blocchi sia lungo le righe sia lungo le colonne.

Come prima cosa organizziamo logicamente *nprocs* processi in una griglia bidimensionale $G_{r,c}$ di dimensione $P \times Q$. Siano P e Q due interi tali che

$$nprocs \leq P \cdot Q \ ,$$

indichiamo con P_i il processore di identificativo i, $0 \leq i < nprocs$ e consideriamo l'applicazione F che ad ogni processore associa un nodo nella griglia $G_{r,c}$:

$$F : i \in \{0, ..., .nprocs - 1\} \rightarrow (r_i, c_i) \in \{0, ..., P-1\} \times \{0, ..., Q-1\}$$

dove

$$r_i = \frac{i}{Q}$$

$$c_i = mod(i, Q)$$

sono le coordinate della posizione del processore P_i in $G_{r,c}$ (Fig. 5.12).

La matrice viene suddivisa in blocchi di dimensione fissata, e tali blocchi vengono poi distribuiti tra i processori della griglia (Fig. 5.13).

Supponiamo per semplicità che i blocchi siano quadrati di dimensione $nb \times nb$; il generico elemento di indici globali (m, n) è memorizzato nella posizione (i, j) del blocco (b, d) nel processore

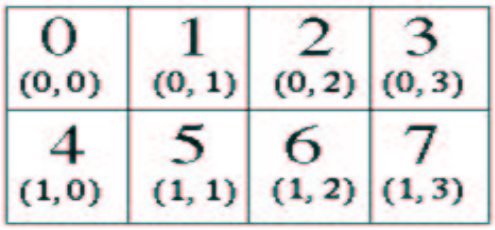


Figura 5.12: *In figura è mostrato come vengono riorganizzati $p = 8$ processi in una griglia bidimensionale.*

(p, q), dove

$$(p, q) = \left(mod\left(\left\lfloor \frac{m}{nb} \right\rfloor, P \right), mod\left(\left\lfloor \frac{n}{nb} \right\rfloor, Q \right) \right)$$

$$(b, d) = \left(\left\lfloor \frac{\left\lfloor \frac{m}{nb} \right\rfloor}{P} \right\rfloor, \left\lfloor \frac{\left\lfloor \frac{n}{nb} \right\rfloor}{Q} \right\rfloor \right)$$

$$(i, j) = (mod(m, nb),\ mod(n, nb))$$

La distribuzione ciclica a blocchi è una generalizzazione della distribuzione ciclica (anche detta *wrapped-around* o *scattered*), e della distribuzione a blocchi:

- la distribuzione a blocchi assegna ai processi blocchi di elementi contigui della matrice globale (Fig. 5.14).

  Ponendo

  $$nb = \left\lceil \frac{m}{P} \right\rceil, \quad nb = \left\lceil \frac{n}{Q} \right\rceil$$

  otteniamo dalla distribuzione ciclica a blocchi la distribuzione a blocchi:

  $$(p, q) = \left(\left\lfloor \frac{m}{nb} \right\rfloor, \left\lfloor \frac{n}{nb} \right\rfloor \right)$$

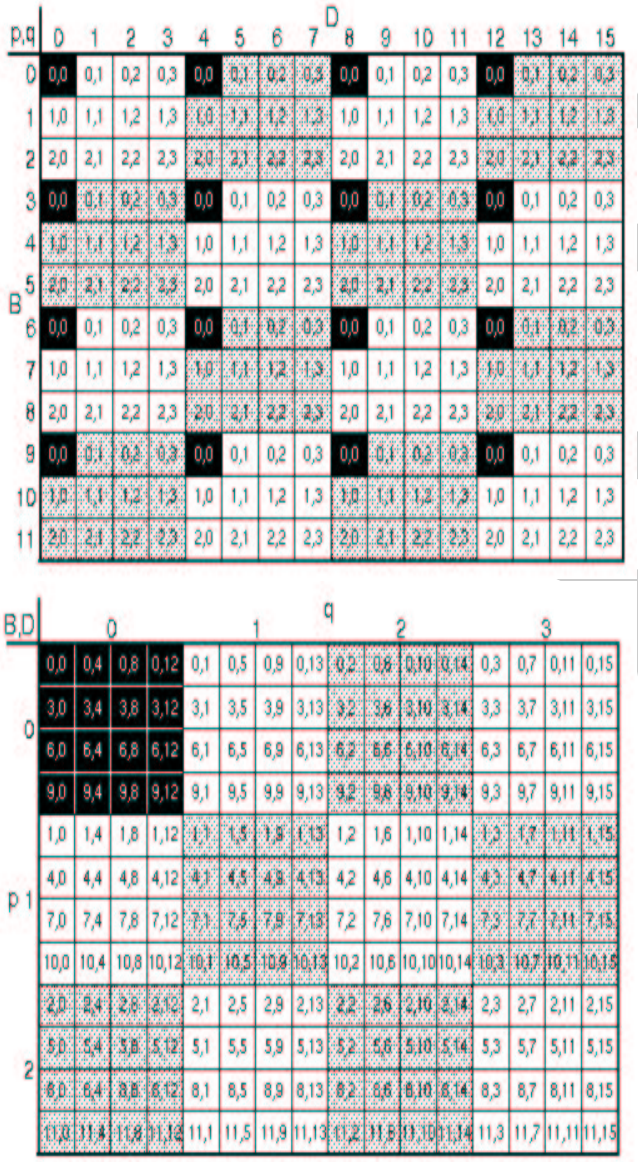


Figura 5.13: *Distribuzione ciclica a blocchi di uma matrice di dimensione* 48×80 *in blocchi di dimensione* 4×5 *in una griglia di processi di dimensione* 3×4. *Ogni rettangolo rappresenta un blocco di matrice (i singoli elementi non sono rappresentati). Gli indici* p, q *individuano il processo a cui viene assegnato il blocco. Gli indici* B, D *individuano il blocco da assegnare. Nella prima figura è rappresentata la distribuzione dei blocchi ai processi. Nella seconda figura vengono evidenziati per ogni processo i blocchi che gli vengono assegnati.*

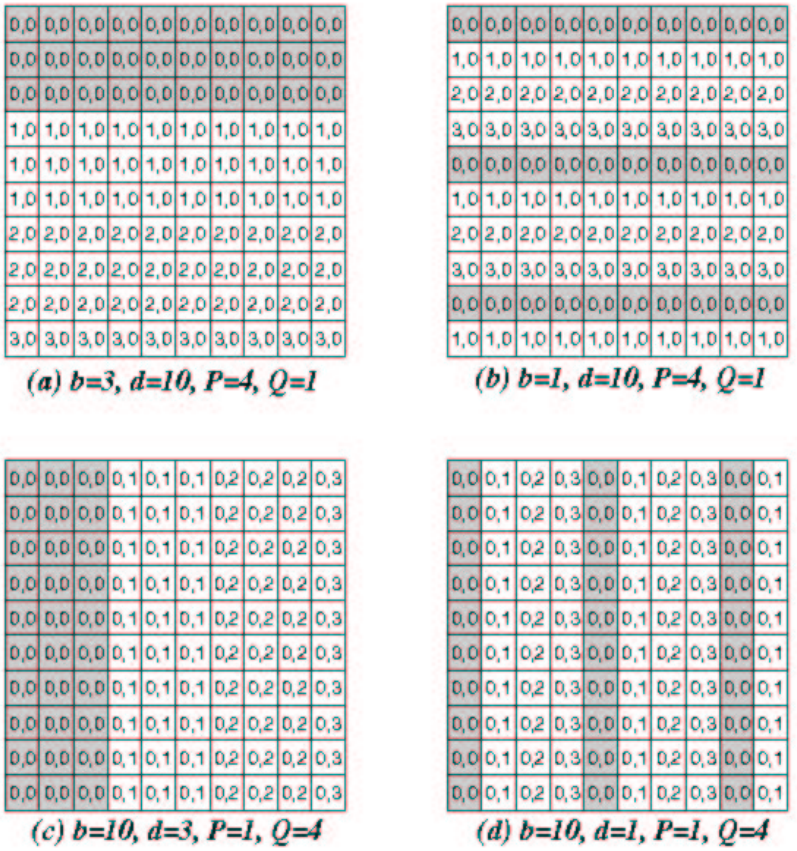


Figura 5.14: *Nelle quattro figure sono mostrate alcune distribuzioni di una matrice di dimensione* 10×10. *Ogni cella rappresenta un elemento della matrice ed è contrassegnato dalla posizione* (p, q) *del processo della griglia a cui viene assegnato. Le figure* (a) *e* (b) *mostrano la distribuzione a blocchi e ciclica per righe su* 4 *processi. Le figure* (c) *e* (d) *mostrano la distribuzione a blocchi e ciclica per colonne su* 4 *processi. Sotto ogni figura è specificata la dimensione del blocco* $(r \times s)$ *e la dimensione della griglia di processi* $(P \times Q)$.

$$(b, d) = (0, 0)$$

$$(i, j) = (mod(m, nb),\ mod(n, nb))$$

- la distribuzione ciclica assegna elementi contigui della matrice globale a processi successivi (Fig. 5.14).
- la distribuzione scattered può essere ritrovata come caso particolare della distribuzione ciclica a blocchi scegliendo $nb = 1$. In questa circostanza

$$(p, q) = (mod(m, P),\ mod(n, Q))$$

$$(b, d) = \left(\left\lfloor \frac{m}{P} \right\rfloor, \left\lfloor \frac{n}{Q} \right\rfloor \right)$$

$$(i, j) = (0, 0)$$

5.3.3 Strategia di parallelizzazione della Fattorizzazione LU

In questo paragrafo descriviamo l'implemetazione dell'algoritmo di fattorizzazione LU con pivoting parziale di una matrice A, distribuita secondo uno schema ciclico a blocchi. Supponiamo che $A \in \Re^{M \times N}$ e suddiviamola in blocchi quadrati di dimensione $nb \times nb$. Allora la matrice A risulta composta da $[M_b \times N_b]$ blocchi con

$$M_b = \left\lceil \frac{M}{nb} \right\rceil \quad N_b = \left\lceil \frac{N}{nb} \right\rceil$$

La fattorizzazione LU consiste in un insieme di passi

$$k = 0, \ldots, min(M_b, N_b) - 1$$

in ognuno dei quali vengono eseguite le tre operazioni seguenti:

1. fattorizzazione LU con pivoting parziale del $k-esimo$ blocco di colonne. Vengono costruite le matrici L_0, L_1 ed U_0 (Fig. 5.11) ed aggiornati gli scambi di righe per il pivot su tutta la matrice;
2. calcolo del $k-esimo$ blocco di righe della matrice U (matrice U_1) mediante risoluzione del sistema multiplo triangolare inferiore $L_0U_1 = C$;
3. aggiornamento della sottomatrice E mediante sostituzione con la matrice $E' = E - L_1U_1$.

Vediamo come vengono effettuate le tre operazioni.

TASK1 Nella prima parte di questo task è coinvolto un solo blocco di colonne distribuito su una sola colonna della griglia dei processi. Al passo k viene trasformata ognuna delle nb colonne del blocco di colonne k. Consideriamo la colonna i-ma del blocco. L'elemento pivot viene individuato come l'elemento di massimo valore assoluto confrontando gli elementi tra la riga di indice $nb \times k + i$ e l'ultima riga. Gli elementi della matrice coinvolti nella ricerca del pivot sono mostrati nella Fig. 5.15.
Una volta trovato l'elemento pivot, il suo valore e la riga a cui appartiene vengono inviati a tutti i processi e viene scambiata la riga $nb \times k + i$ con la riga contenente il pivot. Infine ogni elemento della colonna pivot viene diviso per il pivot.
Nella seconda parte di questo task vengono aggiornati gli spostamenti di righe per i blocchi di matrici che precedono e seguono il blocco di colonne fattorizzato. I processori che contengono

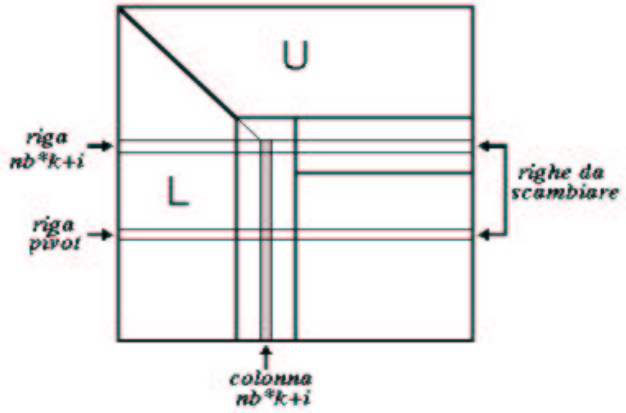


Figura 5.15: *Nella figura è mostrato il pivoting al passo k della fattorizzazione LU. Viene trovto l'elemento piú grande in valore assoluto nella colonna ombreggiata e la riga contenente tale elemento viene scambiata con la riga $nb \times k + i$. Se le righe che devono essere scambiate appartengono a differenti processi sono necessarie delle comunicazioni.*

la riga $nb \times k + i$ e quelli contenente il pivot inviano a tutti i processori della propria riga nella griglia l'indice della riga da scambiare e l'identificativo del processore con cui effettuare lo scambio.

TASK2 La risoluzione del sistema triangolare inferiore $L_0 U_1 = C$ coinvolge un solo blocco di righe distribuito su una sola riga della griglia di processi. La matrice triangolare inferiore L_0 deve essere inviata a tutti i processi che si trovano sulla riga k di processi. Quindi ogni processo della riga k risolve indipendentemente dagli altri il sistema multiplo triangolare inferiore con il blocco C che possiede.

TASK3 Le comunicazioni necessarie per aggiornare la sottomatrice E al passo k hanno luogo in due passi. Prima, ogni processo che possiede una parte della matrice L_1 la invia a tutti i processi che si trovano sulla sua stessa riga della griglia. Questo può

essere fatto contemporaneamente al broadcast della matrice L_0 in modo che tutti i pannelli vengano inviati contemporaneamente. Poi, ogni processo che possiede una parte di U_1 invia questo blocco agli altri processi che si trovano sulla stessa colonna della griglia. A questo punto, ogni processo può completare l'aggiornamento del blocco della matrice E che possiede senza comunicazioni ulteriori.

Nella *Procedura* 5.19 è riportato l'algoritmo per la fattorizzazione LU.
È da notare che la risoluzione del sistema triangolare e l'aggiornamento della matrice operano su blocchi di matrici e possono essere eseguite utilizzando il livello 3 di BLAS (rispettivamente le funzioni *strsm* e *sgemm*). La fattorizzazione della colonna di blocchi, invece, utilizza un ciclo sui blocchi di colonne, quindi può essere eseguita utilizzando le routine di BLAS2. Si può osservare che la fase di aggiornamento della sottomatrice attiva è quella in cui si realizza un parallelismo pieno, ovvero tutti i processori della griglia operano concorrentemente.

Le comunicazioni sono richieste nella fase di ricerca dell'elemento pivot, nello scambio delle righe, e nei broadcast dei blocchi di matrice.

Algoritmo di fattorizzazione LU a blocchi

$pcol = 0$
$prow = 0$
for $i = 1$ **to** $\min(M_b, N_b)$ **do**

for $k = 0$ **to** $r - 1$ **do**
if $(q == pcol)$ **then**
individua il valore del pivot e del suo indice di riga
endif
effettua il broadcast del pivot e del suo indice di riga
scambia la riga corrente con la riga del pivot
if $(q == pcol)$ **then**
dividi gli elementi della colonna r al di sotto della diagonale per il pivot
endif
endfor

if $(p == prow)$ **then**
broadcast L_0 a tutti i processori nello stessa riga p
risolve $L_0 U_1 = C$
endif

if $(q == pcol)$ **then**
ciascun processore effettua il broadcast della propria porzione di L_1
a tutti i processori nella stessa riga
endif
if $(p == prow)$ **then**
ciascun processore effettua il broadcast della propria porzione di U_1
a tutti i processori nella stessa colonna
endif
update $E \leftarrow E - L_1 U_1$

$pcol = (pcol + 1) \bmod Q$
$prow = (prow + 1) \bmod P$
endfor

Procedura 5.19 - Algoritmo per la fattorizzazione LU a blocchi. Nel primo box all'interno del ciclo su k c'è la fattorizzazione della $k-esima$ colonna di blocchi. Nel secondo box viene risolto il sistema multiplo triangolare inferiore per valutare la $i-esima$ riga di blocchi della matrice U. Nel terzo box viene aggiornata la sottomatrice E. (p, q) individuano le coordinate del processo all'interno della griglia.

5.3.4 Analisi dell'efficienza

Analizziamo l'efficienza dell'algoritmo parallelo descritto [11]. Supponiamo che:

- t_{calc} = tempo di esecuzione di un'operazione floating point;
- t_{com} = tempo di comunicazione di un dato f.p. da un processore ad un altro;
- la griglia sia composta da $P = p \times q$ processi ditribuiti su p righe e q colonne;
- nb sia la dimensione dei blocchi;
- N sia l'ordine della matrice A.

Per stimare l'efficienza, definita da

$$E = \frac{T_1}{P \cdot T_P} = \frac{2/3N^3\, t_{calc}}{P \cdot T_P}$$

valutiamo T_P. Richiamiamo le fasi in cui è scandito l'algoritmo:

Per ogni pannello **k=1, ... , N/nb**

TASK1

1. *applica la fatt. LU con pivoting parziale al pannello* $A((k-1)nb+1:N,(k-1)nb+1:k\cdot nb)$ *di dimensione* $(N-(k-1)nb)\times nb$

   La lunghezza dei messaggi in questa fase è trascurabile, per cui il costo di comunicazione predominante è quello di latenza (α) e quindi

   $$t_{com} = \alpha + 2\alpha \log_2(P)$$

il primo addendo è relativo alla latenza dello scambio delle righe contenenti il pivot con la riga $nb \times k + i$, il secondo rappresenta la latenza dei 2 broadcast per l'invio a tutti i processi del valore del pivot e della riga a cui esso appartiene[4].
Si ha:

$$T^1_{calc}(task1) = \left(\frac{2 \cdot nb^3}{3} + (N - k \cdot nb) \cdot nb^2\right) t_{calc}$$

in cui il primo fattore rappresenta il tempo di calcolo delle matrici L_0 e U_0, mentre il secondo è relativo al calcolo della matrice L_1.
Per quanto riguarda il tempo di comunicazione relativo al TASK1 si ha:

$$T^1_{com}(task1) = [\alpha + nb \cdot t_w + 2log_2(p) \cdot (\alpha + t_w) + log_2(p) \cdot (\alpha + nb \cdot t_w)] \cdot n_{scambi}$$

dove:

- $\alpha + nb \cdot t_w$ è il tempo di comunicazione per lo scambio di righe di lunghezza nb;
- $2log_2(p) \cdot (\alpha + t_w)$ è il tempo dei due broadcast del pivot (indice di riga da spedire ed identificativo del processore con cui effettuare lo scambio) che avvengono lungo la colonna della griglia che ha effettuato la fattorizzazione del pannello A;
- $log_2(p) \cdot (\alpha + nb \cdot t_w)$ è il contributo per il broadcast della riga di lunghezza nb lungo la colonna della griglia che ha effettuato la fattorizzazione del pannello A;
- n_{scambi} è il numero totale di scambi effettuati nella fatto-

[4]Per effettuare il broadcast ad albero di un dato f.p., con P processori, occorrono $log_2(P)$ passi.

rizzazione LU del blocco di A e verifica, quindi la relazione $0 \leq n_{scambi} \leq (nb-1)$.

2. *effettua due broadcast delle informazioni sul pivot (indice della riga da spedire ed identificativo del processore con cui comunicare).*

   I broadcast avvengono lungo le due righe della griglia che devono effettuare scambi. Abbiamo quindi

   $$T^2_{com}(task1) = [\ 2log_2(q)(\alpha + t_w)] \cdot n_{scambi}$$

3. *applica gli scambi alle colonne diverse da quelle del pannello fattorizzato*

   I processori sono organizzati in una griglia $p \times q$; ogni pannello contiene nb colonne ed ogni processore della riga che partecipa allo scambio deve comunicare

   $$\left\lceil \frac{N}{q} \right\rceil - nb$$

   elementi. Quindi, in totale lo scambio delle righe costa

   $$T^3_{com}(task1) = n_{scambi} \cdot \left[\alpha + \left(\left\lceil \frac{N}{q} \right\rceil - nb \right) \cdot t_w \right]$$

TASK2

1. *spedisce in broadcast lungo la riga il pannello fattorizzato*

   Ciascun processore della colonna della griglia tra i quali è distribuito il pannello appena fattorizzato spedisce in broadcast lungo la riga il blocco di L in esso memorizzato. Se esso viene

effettuato in pipeline con una topologia ad anello il suo costo è dato da

$$T^1_{com}(task2) = \alpha + \left\lceil \frac{N-(k-1)\cdot nb}{p} \right\rceil nb \cdot t_w$$

osserviamo che il termine $\log_2(q)$ è omesso poiché tale operazione può essere eseguita in pipeline in sovrapposizione al calcolo, pertanto compare solo il tempo di comunicazione speso dal processore sorgente.

2. *calcola il pannello orizzontale di U relativo al passo*

In questa fase abbiamo

$$T^2_{calc}(task2) = \left\lceil \frac{N-(k-1)\cdot nb}{q} \right\rceil nb^2 \cdot t_{calc}$$

tempo di calcolo dovuto alla risoluzione dei sistemi;

$$T^2_{com}(task2) = \log_2(p) \cdot \left[\alpha + \left(\left\lceil \frac{N-(k-1)\cdot nb}{q} \right\rceil nb \right) \cdot t_w \right]$$

tempo di comunicazione dovuto al broadcast lungo le colonne del blocco di U appena calcolato.

TASK3

1. *aggiorna la matrice attiva*

Prodotto matrice per matrice:

$$T^1_{calc}(task3) = \left\lceil \frac{N-(k-1)\cdot nb}{p} \right\rceil \cdot \left\lceil \frac{N-(k-1)\cdot nb}{q} \right\rceil \cdot nb \cdot t_{calc}$$

endfor

Complessivamente, sommando i contributi relativi ad ogni fase e i k passi abbiamo per l'efficienza E una espressione del tipo:

$$E = \left[1 + \frac{P}{N^2}\left(c_1 log_2(p) + c_2\right)\frac{\alpha}{t_{calc}} + \frac{p}{N}\left(c_3 log_2(p)\frac{t_w}{t_{calc}} + c_4\right) + \right.$$
$$\left. + \frac{q}{N}\left(c_5\frac{t_w}{t_{calc}} + c_6\right)\right]^{-1}$$

dove le costanti $c_i,\ \ i = 1, 2, \ldots, 6$ dipendono solo dal fattore di blocco nb.

Supponiamo che la griglia sia strutturata ad anello, precisamente poniamo $p = 1$, $q = P$; per N sufficientemente grande

$$E \simeq \frac{1}{1 + c_2\frac{P}{N^2}\frac{\alpha}{t_{calc}} + \frac{P}{N}\left(c_5\frac{t_w}{t_{calc}} + c_6\right)}$$

in questo caso, cosí come per quello complementare $p = P$, $q = 1$, al crescere di N, per garantire che l'efficienza E non vada a zero, il rapporto $\frac{P}{N}$ deve mantenersi costante, ovvero che sia

$$\lim_{N\to\infty}\frac{P}{N} = cost$$

ciò significa che $P = q$ $(P = p)$ deve crescere linearmente con N. Se invece assumiamo una topologia di connessione a griglia $p \times q$ e supponiamo che il rapporto q/p sia costante, scegliendo ad esempio $p = u\sqrt{P}$ e $q = v\sqrt{P}$ con u e v costanti, allora

$$E \simeq \frac{1}{1 + \frac{P}{N^2}\left(c_1 log_2(p) + c_2\right)\frac{\alpha}{t_{calc}} + \frac{\sqrt{P}}{N^2}\left(c_3 log_2(p)\frac{t_w}{t_{calc}} + c_4 + c_5\frac{t_w}{t_{calc}}\right)}$$

al crescere di N, per garantire che l'efficienza E non vada a zero, il rapporto $\frac{\sqrt{P}=\sqrt{p\times q}}{N^2}$ deve mantenersi costante, ovvero

$$\lim_{N\to\infty}\frac{\sqrt{p \times q}}{N^2} = cost$$

ciò significa che $\sqrt{P}$ deve crescere linearmente con N^2.

In generale, la topologia ottimale della griglia di processi per l'algoritmo di fattorizzazione LU descritto è rettangolare di dimensione $p \times q$ con $p < q$ (griglia *flat*): ciò è dovuto al fatto che i broadcast lungo le righe possono essere effettuati utilizzando una topologia ad anello e la comunicazione può essere sovrapposta al calcolo. D'altra parte, notiamo che, per N sufficientemente grande, dominano gli ultimi 2 addendi aventi N al denominatore, ed il loro valore può essere diminuito scegliendo $P < Q$. Ad esempio, su macchine Intel il rapporto ottimale è $\frac{P}{Q} = \frac{1}{2}$ [3].

La versione di ScaLAPACK della fattorizzazione LU (PDGETRF), molto simile alla versione di LAPACK (DGETRF), è illustrata nella *Procedura* 5.20 .

LAPACK

```
      DO 20 J=1,MIN(M,N),NB
       JB=MIN(MIN(M,N)-J+1,NB)
*
*
* Factor diagonal and subdiagonal blocks
* and test for exact singularity.
*
       CALL DGETF2(M-J+1,JB,A(J,J),LDA,IPIV(J),
$                  IINFO)
*
* Adjust INFO and the pivot indices.
*
       IF(INFO.EQ.0.AND.IINFO.GT.0)
$         INFO=IINFO+J-1
       DO 10 I=J,MIN(M,J+JB-1)
          IPIV( I ) = J - 1 + IPIV( I )
  10   CONTINUE
*
* Apply interchanges to columns 1:J-1.
*
      CALL DLASWP(J-1,A,LDA,J,J+JB-1,IPIV,1)
*
*
      IF(J+JB.LE.N) THEN
*
* Apply interchanges to columns J+JB:N.
*
         CALL DLASWP(N-J-JB+1,A(1,J+JB),LDA,J,
$                    J+JB-1,IPIV,1)
*
*Compute block row of U.
*
         CALL DTRSM('Left','Lower','No transpose','Unit',
  $                 JB,N-J-JB+1,ONE,A(J,J),LDA,A(J,J+JB),
  $                    LDA )
         IF(J+JB.LE.M ) THEN
*
* Update trailing submatrix.
*
           CALL DGEMM('No transpose','No transpose',
  $                 M-J-JB+1,N-J-JB+1,JB,-ONE,A(J+JB,J),
  $                 LDA,A(J,J+JB),LDA,ONE,A(J+JB,J+JB),
  $                 LDA )
         END IF
       END IF
  20   CONTINUE
```

ScaLAPACK

```
      DO 10 J=JA,MIN(M,N)-1,DESCA(4)
       JB=MIN(MIN(M,N)-J+JA,DESCA(4)
*
I=IA+J-JA
* Factor diagonal and subdiagonal blocks
* and test for exact singularity.
*
       CALL PDGETF2(M-J+JA,JB,A,I,J,DESCA,
$                  IPIV,IINFO)
*
* Adjust INFO and the pivot indices.
*
       IF(INFO.EQ.0.AND.IINFO.GT.0)
$         INFO=IINFO+J-JA
*
*
*
*
* Apply interchanges to columns JA:J-JA.
*
      CALL PDLASWP('Forward','Rows',J-JA,A,IA,JA,
$                 DESCA,J,J+JB-1,IPIV)
*
      IF(J-JA+JB.LE.N ) THEN
*
* Apply interchanges to columns J+JB:JA+N-1.
*
         CALL PDLASWP('Forward','Rows',N-J-JB+JA,A,
$                 IA,J+JB,DESCA,J,J+JB-1,IPIV)
*
*Compute block row of U
*
         CALL PDTRSM('Left','Lower','No transpose','Unit
  $                 JB,N-J-JB+JA, ONE,A,I,J,DESCA,A,I,J+J
  $                    DESCA )
         IF(J-JA+JB+1.LE.M ) THEN
*
* Update trailing submatrix.
*
           CALL PDGEMM('No transpose','No transpose',
  $                 M-J-JB+JA,N-J-JB+JA,JB,-ONE,A,I+JB,J
  $                  DESCA,ONE,A,I+JB,J+JB,
  $                 DESCA)
         END IF
       END IF
  10   CONTINUE
```

Procedura 5.20 - Descrizione della routine DGETRF di Lapack e della routine PDGETRF di ScalLAPACK .

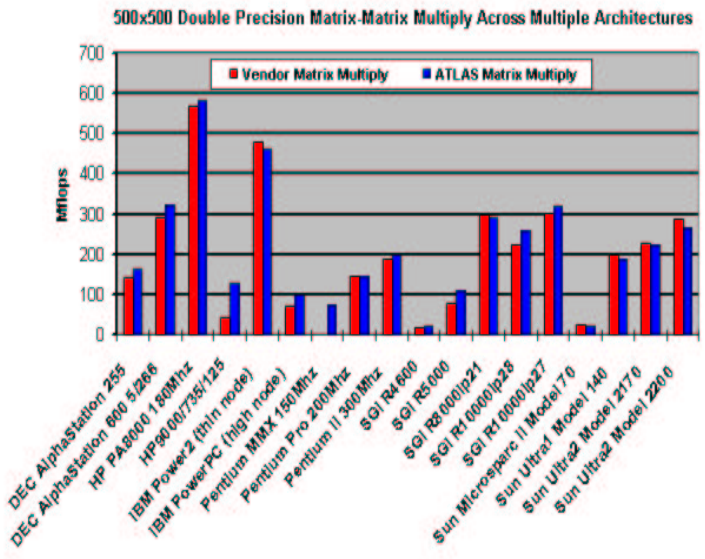


Figura 5.16: *L'operazione matrice per matrice su differenti architetture: confronto tra le librerie ottimizzate fornite dai venditori e ATLAS*

5.4 Cenni ad ATLAS

Essendo BLAS una libreria portabile ed efficiente è comunemente usata nello sviluppo di software per il calcolo matriciale, per cui è necessario avere delle implementazioni ottimizzate. Poiché l'ottimizzazione di BLAS per una specifica architettura necessita di un processo lungo e costoso, si è sviluppata una metodologia per la generazione automatica di routine efficienti per gli attuali microprocessori.

È cosí che nasce **ATLAS** (**A**utomatically **T**uned **L**inear **A**lgebra **S**oftware) sviluppato da R. Clint Whaley e J. Dongarra [54] e capace di eguagliare o addirittura superare le performance raggiunte dai venditori con le versioni ottimizzate di BLAS come mostrato in Fig. 5.16.

ATLAS rappresenta un'applicazione portabile ed efficiente delle metodologie *AEOS* (**A**utomated **E**mpirical **O**ptimization of **S**oftware) alle routine di BLAS. Come ogni implementazione di AEOS, anche

ATLAS necessita alcuni requisiti:

- *Adeguato compilatore ANSI C.*
  ATLAS è interamente scritto in ANSI C con l'eccezione di alcune interfacce in FORTRAN77 che richiamano routine in C. *ATLAS* non richiede un compilatore ottimale, dato che utilizza i generatori di codice che effettuano molte delle ottimizzazioni normalmente fatte dai compilatori (*loop unrolling*[5], *latency hiding*, ecc.).

- *Memoria Gerarchica.*
  ATLAS esige che l'architettura sia dotata di memoria gerarchica, in quanto risultati migliori si ottengono quando sono presenti sia i registri che il livello 1 di cache.

[5]Spesso la parte piú critica di un codice, dal punto di vista del tempo di esecuzione, è costituita dai cicli *for* ... *endfor*, in cui ad esempio viene effettuato il prodotto scalare tra due vettori. Una semplice tecnica per l'ottimizzazione di questi cicli è il *loop unrolling* [8] (*"svolgimento dei cili"*). Quando un ciclo è *"svolto"*, le istruzioni al suo interno sono ripetute piú volte con una riorganizzazione degli indici e del passo di incremento del ciclo. Ad esempio, la routine SAXPY che aggiorna un vettore mediante il prodotto di uno scalare per un vettore

```
    do 10 i=1,n
      y(i)=y(i)+a*x(i)
 10 continue
```

applicando il loop unrolling con un passo 4, assume la forma

```
    res=mod(n,4)
    do 5 i=1,res
      y(i)=y(i)+a*x(i)
 5 continue
    m=n-res
    do 10 i=res+1,m,4
      y(i)=y(i)+a*x(i)
     y(i+1)=y(i+1)+a*x(i+1)
     y(i+2)=y(i+2)+a*x(i+2)
     y(i+3)=y(i+3)+a*x(i+3)
 10 continue
```

In questa riorganizzazione del codice, vengono prima calcolati i mod(n,4) elementi e poi, ad ogni passo, vengono calcolati 4 elementi eseguendo 1 solo accesso ai registri.

"This apparatus was regarded as a prodigy of nature,
because it captured in a machine an art which altogheter
dwells in man's mind, and because thus means were discovered
to execute all operations of this art with perfect confidence
without exerting one's mind. "

Report on the calculator developed by Pascal, 1642

"Questo sistema viene visto come un prodigio della natura,
sia perchè ha (racchiuso) in una macchina un'arte
che è prerogativa della mente umana, sia perchè significa
l'aver scoperto come riprodurre perfettamente tutte le operazioni
di quest'arte senza impiegare la mente umana. "