

1 Parallel GEMVT–Cache Efficiency in Combined Left and Right Matrix Vector Mutli- plcations¹

Gary W. Howell
North Carolina Statue University
gary_howell@ncsu.edu
Charles T. Fulton
Florida Institute of Technology
Madhan Premkumar
Florida Institute of Technology
fulton@zach.fit.edu

2 Abstract

In previous work, combined left and right matrix vector multiplications were used to reduce data transfer in serially computed Householder bidiagonalization. For a matrix A too large to fit in cache memory, performing first a left matrix vector multiply $v^T \leftarrow x^T A$ and then a right matrix vector multiply $y \leftarrow Av$ entails reading A twice from memory into cache. Performing the two matrix vector multiplications in one pass of A through cache, i.e., as one call to `_GEMVT` is more efficient. In the LAPACK Householder bididagonalization, half the flops are BLAS-2 `_GEMV` calls (matrix vector multiplications) and half are BLAS-3. Rarranging the algorithm allows the `_GEMV` calls to be converted to BLAS 2.5 `_GEMVT`, allowing bidiagonalization in significantly less time.

This paper explores the efficient parallel implementation of `_GEMVT`, The effectiveness of parallel `_GEMVT` depends on the architecture. For parallel architectures with large caches and low latency communication, `_GEMVT` requires less time than performing two `_GEMV` calls. We give a simple model to predict the best data layout and show that it agrees with timing runs. We also discuss parallel implementation of the related `_GEMVER` operation. The simple model given here allows prediction of the best distribution for a given architecture.

3 Introduction

Current computer architectures store data in a hierarchy of distances from the computational registers. Data in a small number of registers can be used for computations in the current clock cycle. Data in several levels of cache memory is available in at most a few clock cycles. Accessing data in main memory requires several dozen clock cycles and is constrained by bus bandwidth. Matrices larger than a few hundred square are typically too large to fit in cache memory and must be stored in main storage (RAM). In this paper, we discuss the use of BLAS 2.5 operators to halve data transfer for operations which require two

coupled matrix-vector multiplications. In computers with cache architectures, halving data transfer substantially reduces computational time.

The two BLAS 2.5 routines of interest in this paper are `_GEMVER`, which performs the operations:

$$\begin{aligned}\hat{A} &\leftarrow A + u_1 v_1^T + u_2 v_2^T \\ x &\leftarrow \beta \hat{A}^T y + z \\ w &\leftarrow \alpha \hat{A} x\end{aligned}$$

and `_GEMVT`, which performs the operations:

$$\begin{aligned}x &\leftarrow \beta A^T y + z \\ w &\leftarrow \alpha A x\end{aligned}$$

Specifications for these two routines are part of the new BLAS standard [?].

4 Blocked GEMVT Operator

We found it efficient to implement `_GEMVT` on top of the BLAS-2 `_GEMV` operator as follows. Partition a general $m \times n$ rectangular matrix A as

$$[A_1 \quad A_2 \quad \dots \quad A_j] \quad (1)$$

Then `_GEMVT` is performed as follows:

`_GEMVT`

```

For  $i = 1, j$ 
     $x_i^T \leftarrow \beta y^T A_i$            Call to _GEMV
     $w \leftarrow w + \alpha A_i x_i$      Another call to GEMV
End for

```

where k is the number of columns in each block A_i and, for simplicity, we assume $k * j = n$. This version of `_GEMVT` loops only once through the columns of A , with each successive block of columns of A read into cache and used in two calls to `_GEMV`.

The time for serial `_GEMVT` can be modelled by considering the time for data movement from RAM to cache. As a simple model, take the time to move data from RAM to cache as

$$Time_{for_communication} = Time_{to_start} + (number_of_bytes_in_message) * (Time_{to_transmit_another_byte})$$

so that we have the time to transmit a message to cache as

$$TC_{cache} = TS_{cache} + L_{cache} B_{cache}.$$

Taking times in terms of “equivalent flops”, we typically have

$$TS_{cache} = 100 + (\text{number_of_double_precision_numbers})16.$$

Performing a matrix-vector multiplication (`_GEMV`) for an $m \times n$ matrix not in cache requires $2mn$ flops. The total time (in peak flops) for computation is (neglecting the latency TS_{cache} as may be appropriate when data is fetched in the order it is stored).

$$2mn + 16mn$$

so that the efficiency as a fraction of peak speed is

$$\frac{\text{time_for_flops}}{\text{time_for_GEMV}} = \frac{2mn}{2mn + 16mn} = \frac{1}{9}$$

Performing `_GEMVT` for matrices too large to fit in cache gives this same efficiency. But if `_GEMVT` can load data only once, then $4mn$ flops are performed with the same time for loading data so that the efficiency as a fraction of peak speed is

$$\frac{\text{time_for_flops}}{\text{time_for_GEMVT}} = \frac{4mn}{4mn + 16mn} = \frac{1}{5}$$

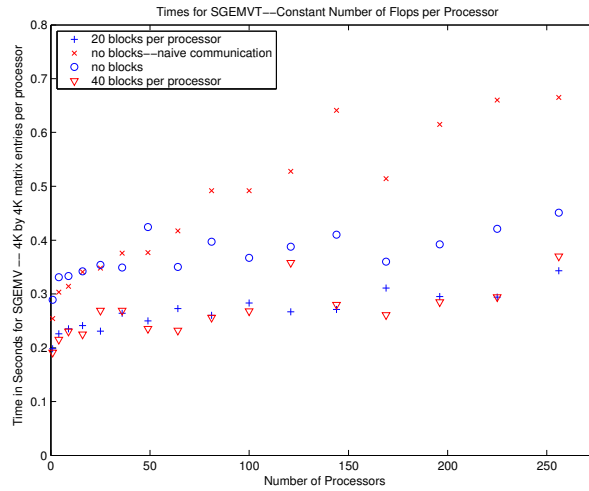


Figure 1: These runs were with on an ALPHA SC, 8 MByte cache, Quadrics interconnect

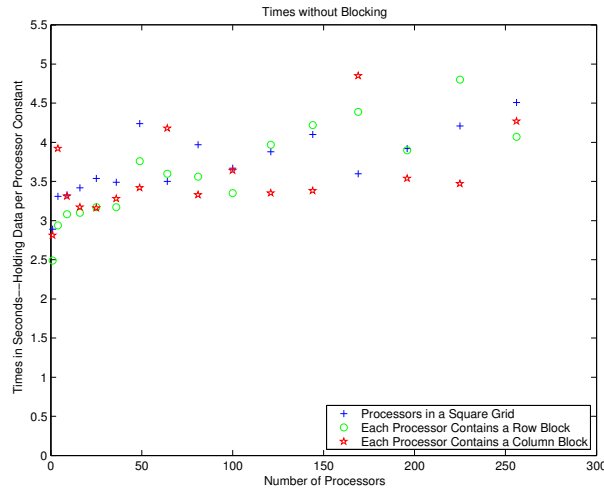


Figure 2: These runs were with on an ALPHA SC, 8 MByte cache, Quadrics interconnect. Comparing with the last plot shows the effect of blocking for gemvt on a given processor.

5 Analysis of Parallel GEMVT Codes

We consider the distributed memory parallel case. In a distributed memory case, each element of A will be stored in memory on one processor. In considering parallel efficiency, we are considering $_GEMVT$ as part of the n row-column eliminations required for bidiagonalization. We want $_GEMVT$ to be scalably efficient and to draw conclusions to the scalability of Householder bidiagonalization.

We suppose that we have control over the initial distribution of A . Since the total cost of bidiagonalization is $O(n^3)$, then the cost of redistributing A may in comparison be negligible. Since n $_GEMVT$ operations are performed, we are justified in neglecting the initial cost of distributing nm elements of A . We need only consider communications required to multiply A by x and y and to distribute such parts of x and y as to start the next $_GEMVT$ operation.

In order to model cache efficiency in parallel computations, we need to model communication times between RAM and cache and also communication times between processors. We use the same simple model for each.

$$Time_{for_communication} = Time_{to_start} + (number_of_bytes_in_message) * (Time_{to_transmit_a_word_byte})$$

so that we have the time to transmit a message to cache as

$$TC_{cache} = TS_{cache} + L_{cache}B_{cache}.$$

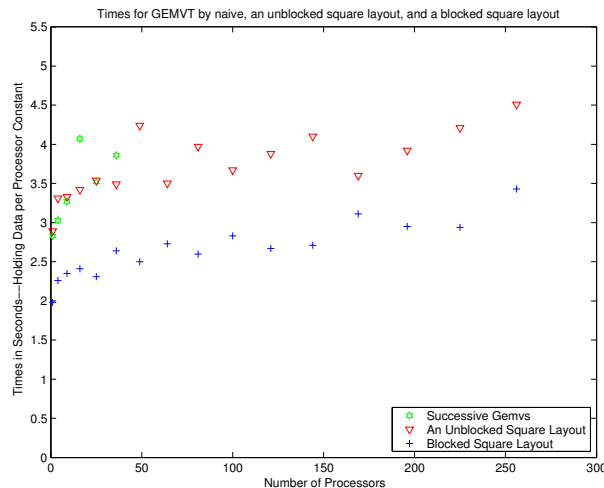


Figure 3: These runs were with on an ALPHA SC, 8 MByte cache, Quadrics interconnect. These show the effect of blocking for gemvt on a given processor.

and the time to communicate a message between nodes is

$$TC_{node} + TS_{node} + L_{node}B_{node}.$$

A convenient unit is “equivalent flops”, i.e., how many floating point operations the CPU working at peak speed could accomplish in the same time. The architectures on which we have the most current experience are an SC45 with a quadrics interconnect and a Pentium III with a fast ethernet interconnect.

For the SC45, we have approximately

$$TC_{cache} = 100 + L_{cache} \quad \text{and} \quad TC_{node} = 1.e4 + L_{node}20$$

where L_{cache} is limited by the $8.e6$ byte capacity of the L2 cache. The time to load the entire cache is around 8 million flops, comparable to the time to start 800 or a thousand messages.

For the Pentium III, we have approximately

$$TC_{cache} = 100 + L_{cache} \quad \text{and} \quad TC_{node} = 1.e5 + L_{node}100$$

where L_{cache} is limited by the $256e3$ byte capacity of the L2 cache. The time to load the cache is around 256K flops, comparable to the 100K flops required to start a message.

The “800 messages started per cache reload” property of the SC45 allows the `_GEMVT` operation to use cache efficiently. For the Pentium III with a fast ethernet connection, the time for communicating between processors is long enough that reuse of cached data is less effective.

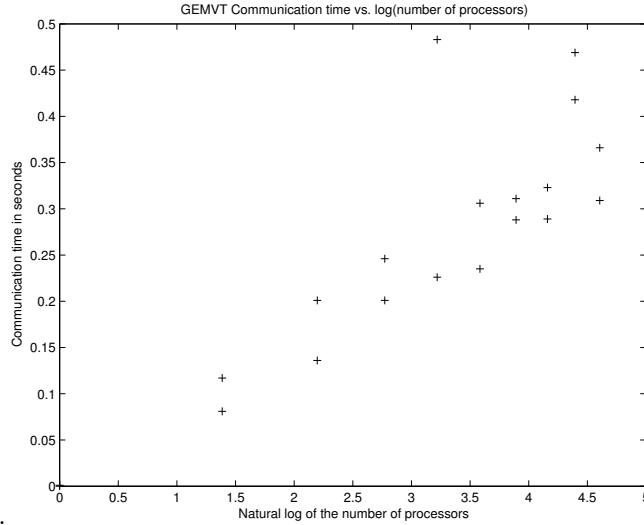


Figure 4: These runs were with on an ALPHA SC, 8 MByte cache, Quadrics interconnect. These leave out the computation, showing that the time for communication grows like the log of the number of processors.

Below, we'll see that distributing A in other ways is usually preferable. But in order to have a definite example to model let's consider distributing A by columns, i.e., partitioning $A = [A_1|A_2|\dots|A_p]$ so that each processor j contains the partition A_j consisting of some columns of A .

$$[A_1 \quad A_2 \quad \dots \quad A_j] \tag{2}$$

Then `_GEMVT` is performed as follows:

`_GEMVT`

```
In parallel  $i = 1, p$ 
     $x_i^T \leftarrow \beta y^T A_i$            Call to _GEMV
     $w_i \leftarrow \alpha A_i x_i$        Another call to GEMV
End Parallel AllReduce on  $w_i$  producing  $y = \sum_i w_i$  on all processors
```

In order to re-use in-cache data on each processor, subblocking is used so that each processor performs a loop over j

$$x_{i_j}^T \leftarrow \beta y^T A_{i_j}$$

$$w_i \leftarrow \alpha A_{i_j} x_{i_j} + w_i$$

As the number of rows of A becomes large, the number of columns that will fit in cache grows small so that A_{i_j} becomes only one or two columns fit in cache.

For the case that $A_{i,j}$ is a single column, the loop on each processor becomes an inner product of the column with x , followed by an `_axpy` operation, requiring for cache efficiency that the three vector $A_{i,j}$, x , and w_i remain in cache. For the 256K cache of the Pentium III, at most 32K double precision numbers will fit in cache so that the maximal number of rows that will fit in cache is at most about 10K.

In this formulation, the only communication is in the collective Allreduce for adding up the vectors w_i and depositing the sum x on each processor. Here the vectors w_i and x are each of length m , which is the number of rows of A . The optimal algorithm for performing an Allreduce is to perform pairwise adds of the w_i vectors and a fan-out broadcast of the x vector. Both the fan-in and fan-out require $\log_2(p)$ parallel stages. The estimated single precision time for the Allreduce is then $2 \log_2(p)$ times the time to pass one vector of length m , i.e.,

$$2 \log_2(p)(TS_{node} + B_{node}8m).$$

flops.

When columns are short enough to fit in cache, the computational time for p processors is (neglecting the latency TS_{cache})

$$\frac{4mn}{p} + \frac{mn}{p}8B_{cache}$$

where B_{cache} is the number of flops which would be performed in the time for one additional byte to be transmitted. Typically $B_{cache} \approx 2$ so that the peak speed for a BLAS-2 matrix-vector multiplication is about half the processor peak speed. If so that the peak rate for a BLAS-2 computation for a total communication and computational time of

$$2 \log_2(p)(TS_{mode} + B_{mode}8m) + \frac{4mn}{p} + \frac{mn}{p}8B_{cache} \quad (3)$$

Else if the matrix columns are too long to fit in cache, then data transfer in a node doubles, giving a total time of

$$2 \log_2(p)(TS_{mode} + B_{mode}8m) + \frac{4mn}{p} + 2\frac{mn}{p}8B_{cache} \quad (4)$$

For either Equation ?? or ??, $\frac{mn}{p}8B_{cache}$ is usually the predominant term.

/sectionSome numerical experiments
As an initial experiment, we did some runs with the following version of parallel dgemvt. For these runs, we held the amount of data per processor constant. As predicted, communication costs go as the log of the number of processors used. On these clusters the latency of the quadric switch is on the order of a few microseconds. Reloading the 8 Mbyte cache requires on the order of a few milliseconds. Thus performing a matrix vector multiply on a block of data, waiting for an MPI-ALLReduce, and then doing another matrix multiply is comparatively faster than doing a matrix vector multiply, reading in a new block of data and doing another matrix vector multiply.

For machines with smaller caches and slower interconnects, the re-use of incache data may be less efficient than reloading the cache with new data (and not having to wait).

6 The code

```
c REVISÉD GARY HOWELL--to do parallel gemvt -- October 6, 2003.
C
C Input:  matrix A, vector x, currently hardwired.
C Output: results of calls to various functions testing topology
C        creation
C
C Algorithm:
C   1.  Build a 2-dimensional Cartesian communicator from
C       MPI_Comm_world
C   2.  Print topology information for each process
C   3.  Use MPI_Cart_sub to build a communicator for each
C       row of the Cartesian communicator
C   4.  Carry out a broadcast across each row communicator
C   5.  Use MPI_Cart_sub to build a communicator for each
C       column of the Cartesian communicator
C   6.  Carry out a broadcast across each column communicator
C   7.  Initialize a local matrix A on each processor
C       and a local x vector
C   8.  Perform parallel gemvt by
C       Obtaining local w via a loop on local column blocks
C         a.  column blocking local A, performing  $x^T A_i$ 
C         b.  reduce scatter (on column communicator)
C             to get a bit of local y ( $y_i$ )
C         c.  Perform  $w = A_i y_i + w$ 
C       End Loop
C       Perform a Reduce Scatter in the row communicator,
C         summing up the local w's to get the new local x.
C
C Limitations of code so far:
C   1) each processor has to have the same size matrix
C   2) block size kb is hardwired at 100
C   3) the number of processors total must be a perfect square.
C
C Note: Assumes the number of process, p, is a perfect square
C
C See Chap 7, pp. 121 \& ff in PPMPI
C
C These chapters refer to Peter Pacheco's book on MPI which
```

c (Morgan Kaufman Press) which
c is a good introduction to parallel computing and the
c MPI library. Some of the variables here are inherited
c from one of his sample codes using subcommunicators.
C

```
PROGRAM PGEMVT
INCLUDE 'mpif.h'
integer lda
parameter (lda=5000)
integer      p,m,n,k,nb,i,j
real        preal
real        a(lda,lda),x(lda),y(lda),z(lda),w(lda)
integer      my\_rank
integer      q
integer      grid\_comm
integer      dim\_sizes(0:1)
integer      wrap\_around(0:1)
integer      reorder
integer      coordinates(0:1)
integer      my\_grid\_rank
integer      grid\_rank
integer      free\_coords(0:1)
integer      row\_comm
integer      col\_comm
integer      row\_test
integer      col\_test
integer      ierr
real*8      entim,start
```

C
C

```
m = 2000
n = 2000
k = 10
reorder = 1
call MPI\_INIT( ierr)
call MPI\_COMM\_SIZE(MPI\_COMM\_WORLD, p, ierr )
call MPI\_COMM\_RANK(MPI\_COMM\_WORLD, my\_rank, ierr )
```

C

```
preal = p
q = sqrt(preal)
```

C

```
dim\_sizes(0) = q
dim\_sizes(1) = q
wrap\_around(0) = 0
wrap\_around(1) = 0
call MPI\_CART\_CREATE(MPI\_COMM\_WORLD, 2, dim\_sizes,
```

```

+      wrap\_around, reorder, grid\_comm, ierr)
C
  call MPI\_COMM\_RANK(grid\_comm, my\_grid\_rank, ierr)
  call MPI\_CART\_COORDS(grid\_comm, my\_grid\_rank, 2,
+      coordinates, ierr)
C
  call MPI\_CART\_RANK(grid\_comm, coordinates, grid\_rank,
+      ierr)
C
  free\_coords(0) = 0
  free\_coords(1) = 1
  call MPI\_COMM\_SPLIT(MPI\_COMM\_WORLD,coordinates(0),my\_rank,
+      row\_comm,ierr)
  if (coordinates(1) .EQ. 0)then
    row\_test = coordinates(0)
  else
    row\_test = -1
  endif
  call MPI\_BCAST(row\_test, 1,MPI\_INTEGER, 0,row\_comm, ierr)
  free\_coords(0) = 1
  free\_coords(1) = 0
  call MPI\_BARRIER(MPI\_COMM\_WORLD,ierr)
  call MPI\_COMM\_SPLIT(MPI\_COMM\_WORLD,coordinates(1),my\_rank,
+      col\_comm,ierr)
  if (coordinates(0) .EQ. 0)then
    col\_test = coordinates(1)
  else
    col\_test = -1
  endif
  call MPI\_BCAST( col\_test, 1, MPI\_INTEGER,0, col\_comm,ierr)
c
c A more general matrix input would be useful
c
c The global matrix has entries  $a(i,j) = i-j$ 
c the global x vector has entrix  $x(i) = i$ 
c   print*, 'my\_rank,9 and 1',my\_rank,coordinates(0),coordinates(1)
c   do j=1,n
c     do i=1,m
c       a(i,j) = coordinates(0)*m+i - (coordinates(1)*n+j)
c     end do
c     y(i) = 0.0
c   end do
c   do i=1,m
c     x(i) = coordinates(0)*m + i
c     w(i) = 0.0
c   end do

```

```

        if(my\_rank.eq.0) then
            start = MPI\_Wtime()
        endif
        do ii=1,10
c Each processor belongs to a column communicator, so will use that
c communicator to help with the global matrix vector product  $x^T A_i$ 
c looping through the column blocks  $A_i$  for the local matrix A
            kb = n/k
            do i=1,m
                x(i) = coordinates(0)*m + i + ii/10.
                w(i) = 0.0
            end do
            do i=1,n/kb
                call sgemv('T',m,kb,1.0,a(1,(i-1)*kb+1),
+                    lda,x,1,1.0,y((i-1)*kb+1),1 )
                call MPI\_ALLREDUCE(y((i-1)*kb+1),z((i-1)*kb+1),
+                    kb,MPI\_REAL,MPI\_SUM,col\_comm,ierr)
c note, dependng on flavor of MPI, may need a barrier here ..
                call sgemv('N',m,kb,1.0,a(1,(i-1)*kb+1),
+                    lda,z((i-1)*kb+1),1,1.0,w,1)
            end do
c Take care of extra columns (for the case that k does not divide n exactly
            call sgemv('T',m,n-(n/kb)*kb,1.0,a(1,(i-1)*kb+1),
+                    lda,x,1,1.0,y((i-1)*kb+1),1 )
            call MPI\_ALLREDUCE(y((i-1)*kb+1),z((i-1)*kb+1),
+                    n-(n/kb),MPI\_REAL,MPI\_SUM,col\_comm,ierr)
            call sgemv('N',m,n-(n/kb)*kb,1.0,a(1,(i-1)*kb+1),
+                    lda,z((i-1)*kb+1),1,1.0,w,1)
c Column communicator operations are done, next do the row communicator
c operation to sum up the local  $Aw$ 's to get the global  $x$ 
            call MPI\_BARRIER(MPI\_COMM\_WORLD,ierr)

            call MPI\_ALLREDUCE(w,x,m,
+                    MPI\_REAL,MPI\_SUM,row\_comm)
            call MPI\_BARRIER(MPI\_COMM\_WORLD,ierr)
c After the scatter reduce, each row communicator has the necessary local  $x$ 
c to start over (or to do local updates)
        end do
        call MPI\_BARRIER(MPI\_COMM\_WORLD,ierr)
        if(my\_rank.eq.0)then
            entim = MPI\_Wtime()-start
            print*,entim
        endif
        call MPI\_FINALIZE(ierr)
    end

```

References

- [1] E. ANDERSEN, Z. BAI, C. BISCHOF, J. DEMMEL, J. DONGARRA, A. GREENBAUM, S. HAMMARLING, A. MCKENNEY, S. OSTROUCHEV, D. SORENSEN, *LAPACK User's Guide*, 3rd. Ed. 1999 SIAM, Philadelphia.
- [2] The web site is <http://olin.fit.edu/beowulf>.
- [3] BLAS TECHNICAL FORUM, www.netlib.org/utk/papers/blast-forum.html, 1999.
- [4] J. CHOI, J. DONGARRA, S. OSTROUCHOV, A. PETITET, D. WALKER, AND R.C. WHALEY, *A proposal for a set of parallel basic linear algebra subprograms*, Computer Science Dept. Tech. Report CS-95-292, University of Tennessee, Knoxville, TN, May 1995. (Also LAPACK Working Note #100).
- [5] J. DONGARRA, S. HAMMARLING, AND D. SORENSEN, Block reduction of matrices to condensed forms for eigenvalue computations. *J. Comput. Appl. Math.* 27:215–227, 1989.
- [6] G. HOWELL, C. FULTON, J. DEMMEL, S. HAMMARLING, AND K. MARMOL, Cache Efficient Bidiagonalization Using BLAS 2.5 Operators, submitted to ACM Trans. on Math. Software (available from <http://my.fit.edu/~cfulton/bidiag.html>)
- [7] G. GOLUB AND W. KAHAN, *Calculating the Singular Values and Pseudo-Inverse of a Matrix*, *SIAM J. Num. Anal.* 2:205–24, 1965.
- [8] S. MALHOTRA, *Parallelization of BLAS 2.5 Operator GEMVT*, M.S. Thesis 2003, Computer Science, Florida Institute of Technology, Melbourne, Florida.
- [9] C. MOLER, *Matrix Computation on Distributed Memory Multiprocessors*, (Ed. M. Heath), SIAM, Philadelphia (1986), 181-195.
- [10] K.S. STANLEY, *Execution Time of Symmetric Eigensolvers*, Ph.D. dissertation, 1997, CS, University of California, Berkeley.
- [11] C. WHALEY AND J. DONGARRA, *Automatically Tuned Linear Algebra Software Proceedings of the Ninth SIAM Conference on Parallel Processing for Scientific Computing*, 1999, available on CD-ROM from SIAM.