Communication Efficient Gaussian Elimination with Partial Pivoting using a Shape Morphing Data Layout

Grey Ballard
James Demmel
Benjamin Lipshitz
Oded Schwartz
Sivan Toledo

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2013-12
http://www.eecs.berkeley.edu/Pubs/TechRpts/2013/EECS-2013-12.html

February 21, 2013

Report Documentation Page

Form Approved
OMB No. 0704-0188

1. REPORT DATE	2. REPORT TYPE	3. DATES COVERED
21 FEB 2013		**00-00-2013 to 00-00-2013**

4. TITLE AND SUBTITLE	5a. CONTRACT NUMBER
Communication Efficient Gaussian Elimination with Partial Pivoting using a Shape Morphing Data Layout	5b. GRANT NUMBER
	5c. PROGRAM ELEMENT NUMBER
6. AUTHOR(S)	5d. PROJECT NUMBER
	5e. TASK NUMBER
	5f. WORK UNIT NUMBER
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) **University of California at Berkeley,Electrical Engineering and Computer Sciences,Berkeley,CA,94720**	8. PERFORMING ORGANIZATION REPORT NUMBER
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)	10. SPONSOR/MONITOR'S ACRONYM(S)
	11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release; distribution unlimited

13. SUPPLEMENTARY NOTES

14. ABSTRACT

High performance for numerical linear algebra often comes at the expense of stability. Computing the LU decomposition of a matrix via Gaussian Elimination can be organized so that the computation involves regular and efficient data access. However, maintaining numerical stability via partial pivoting involves row interchanges that lead to inefficient data access patterns. To optimize communication efficiency throughout the memory hierarchy we confront two seemingly contradictory requirements: partial pivoting is efficient with column-major layout, whereas a recursive layout is optimal for the rest of the computation. We resolve this by introducing a shape morphing procedure that dynamically matches the layout to the computation throughout the algorithm, and show that Gaussian Elimination with partial pivoting can be performed in a communication efficient and cache-oblivious way. Our technique extends to QR decomposition, where computing Householder vectors prefers a different data layout than the rest of the computation.

15. SUBJECT TERMS

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT **unclassified**	b. ABSTRACT **unclassified**	c. THIS PAGE **unclassified**	**Same as Report (SAR)**	**18**	

Acknowledgement

We acknowledge funding from Microsoft (Award #024263) and Intel (Award #024894), and matching funding by U.C. Discovery (Award #DIG07-10227). Additional support comes from ParLab affiliates National Instruments, Nokia, NVIDIA, Oracle and Samsung, as well as MathWorks. Research is also supported by DOE grants DE-SC0004938, DE-SC0005136, DE-SC0003959, DE-SC0008700, and AC02-05CH11231, and DARPA grant HR0011-12-2-0016, and grant 1045/09 from the Israel Science Foundation (founded by the Israel Academy of Sciences and Humanities), and grant 2010231 from the US-Israel Bi-National Science Foundation.

Communication Efficient Gaussian Elimination with Partial Pivoting using a Shape Morphing Data Layout*

Grey Ballard[1], James Demmel[1], Benjamin Lipshitz[1], Oded Schwartz[1], and Sivan Toledo[2]

[1]University of California at Berkeley
[2]Tel-Aviv University

Regular Submission

Abstract

High performance for numerical linear algebra often comes at the expense of stability. Computing the LU decomposition of a matrix via Gaussian Elimination can be organized so that the computation involves regular and efficient data access. However, maintaining numerical stability via partial pivoting involves row interchanges that lead to inefficient data access patterns. To optimize communication efficiency throughout the memory hierarchy we confront two seemingly contradictory requirements: partial pivoting is efficient with column-major layout, whereas a recursive layout is optimal for the rest of the computation. We resolve this by introducing a shape morphing procedure that dynamically matches the layout to the computation throughout the algorithm, and show that Gaussian Elimination with partial pivoting can be performed in a communication efficient and cache-oblivious way. Our technique extends to QR decomposition, where computing Householder vectors prefers a different data layout than the rest of the computation.

*We acknowledge funding from Microsoft (Award #024263) and Intel (Award #024894), and matching funding by U.C. Discovery (Award #DIG07-10227). Additional support comes from ParLab affiliates National Instruments, Nokia, NVIDIA, Oracle and Samsung, as well as MathWorks. Research is also supported by DOE grants DE-SC0004938, DE-SC0005136, DE-SC0003959, DE-SC0008700, and AC02-05CH11231, and DARPA grant HR0011-12-2-0016, and grant 1045/09 from the Israel Science Foundation (founded by the Israel Academy of Sciences and Humanities), and grant 2010231 from the US-Israel Bi-National Science Foundation.

1 Introduction

Do we need to trade off numerical stability for high performance? This has been the most important question in numerical linear algebra for at least 20 years. It has motivated an enormous body of deep research. In this paper we show that for one very famous computation in numerical linear algebra, the answer is no: Gaussian Elimination with partial pivoting can be performed in a communication avoiding way.

High performance computers do not resemble simple computational models like the RAM model. They rely on parallelism and complex memory hierarchies to deliver high performance. In the past, such architectures were confined to supercomputers, but today they are ubiquitous. To run fast, an algorithm must be able to utilize many processors concurrently and to avoid communication as much as possible.

Out of all the effective algorithms for a given problem, only a subset exhibits high levels of parallelism and requires little communication between processors and/or between levels of the memory hierarchy. Does this subset always contain algorithms that are as stable as the best ones for the problem, or do we need to trade off stability for high performance? Consider Csanky's algorithm for matrix inversion: it has long been a classic example of a highly parallel but highly unstable algorithm; no stable algorithm is as parallel. Twenty years ago, one of the authors suggested in an influential paper that even in practice, we must trade off stability in return for useful amounts of parallelism [5]. That paper has motivated a huge amount of research, with two main focal points. One has been the stability of so-called fast (Strassen-like) algorithms; this research has so far culminated in algorithms that are stable and fast in theory, but it remains to be seen whether they are also fast in practice [4]. The other focal point has been in algorithms that perform as little communication as possible, culminating in the definition of communication avoidance [3] and in a class of algorithms with that property.

An algorithm is called *communication avoiding* if it performs asymptotically as little communication as possible in two metrics: the total volume of data measured in words transferred between processors or levels of the memory hierarchy (the *bandwidth* it consumes), and the number of messages or block-transfers that carry this data (and therefore the number of times the message or cache-miss *latency* impacts the execution). To show that an algorithm is communication avoiding, one must exhibit a communication lower bound. For many matrix algorithms, lower bounds of the form $\Omega\left(f/\sqrt{M}\right)$ have been established on the total volume of communication and $\Omega(f/M^{1.5})$ on the number of messages, where f is the number of arithmetic operations performed by the algorithm and M is the size of the fast memory in a hierarchy or the local memory in a distributed memory parallel computer [3, 14].

Minimizing the volume of communication while preserving numerical stability has proved relatively easy for many problems. For Gaussian Elimination with partial pivoting (using the largest-magnitude element in a column to eliminate the rest of the column), a 1997 algorithm with a recursive schedule did the trick [13, 17] for the sequential (memory-hierarchy) case; this algorithm is also cache oblivious, in the sense that its schedule does not depend on M.

Minimizing the number of block-transfers while maintaining stability has proved much harder. The first communication avoiding algorithm for Gaussian Elimination [11] used a pivoting rule called tournament pivoting that was both more complicated and less stable than partial pivoting. A second-generation communication avoiding Gaussian Elimination algorithm [15] was even more complicated, but also more stable. The fundamental challenge that required the new pivoting rules is that partial pivoting steps works well when the matrix is stored by column, whereas updating the reduced matrix works well when the matrix is stored with contiguous blocks. The question of whether the simple, elegant, and stable partial pivoting rule can be used in a communication avoiding algorithm remained open.

In this paper we answer this question in the affirmative for the sequential case using a technique we call *shape morphing*: switching the data layout of parts of the matrix back and forth between column-major layout and recursive block-contiguous layout. Doing so allows Gaussian Elimination to access contiguous

memory locations both when searching for a pivot down a column and applying row interchanges, and when computing the U factor and updating the reduced matrix (Schur complement). The shape morphing steps add data movement overhead to the algorithm, but we show that the overall algorithm remains asymptotically efficient. The algorithm is recursive and also cache-oblivious.

The same technique also produces communication avoiding algorithms for the related problem of QR factorization. In addition, we present a communication efficient algorithm for solving a triangular system where the right sides form a rectangular matrix. This subroutine is necessary inside SMLU, but is also used in several other contexts.

Algorithm 1 SMLU, in words. See Figure 1 and Algorithm 8 for further details.

if one column **then**
    solve the problem for a column
end if
recursively factor the left half
forward permute
reshape everything to recursive format
update right half with triangular solve and Schur update
reshape everything back to column format
recursively factor the right half
back permute
combine pivots

Algorithm 2 SMQR, in words. See Figure 2 for further details.

if one column **then**
    solve the problem for a column
end if
recursively factor the left half
reshape everything to recursive format
update right half with triangular and general matrix multiplies
reshape right half back to column format
recursively factor the right half
reshape right half to recursive format
compute auxiliary triangular matrix T with triangular and general matrix multiplies
reshape everything back to column format

2 Machine Model

We model a sequential computer as having an infinite slow memory and a finite fast memory of size M. All computation takes place in the fast memory, and we consider communication between the fast and slow memory. We count both the number of words of data W (or *bandwidth cost*) and the number of messages S (*latency cost*) transferred, and model the communication time as

$$\alpha \cdot S + \beta \cdot W,$$

where α and β are machine-dependent parameters. There is one more parameter, L, which is the size of the maximum allowed message (or block-transfer size). We make no assumptions on the size of L beyond the trivial requirements $1 \leq L \leq M$.

It is instructive to contrast our model to the ideal-cache model of [10]. There, the authors make a "tall cache" assumption that $M = \Omega(L^2)$. We do not make this assumption, so latency optimality is a stricter requirement in our model. Additionally, their model only allows messages of size L, which is equivalent to setting $\beta = 0$ in our model.

One may also consider models where there is a hierarchy of memories, each faster and smaller than the previous one, where the largest/slowest memory is infinite and the computation occurs only in the smallest/fastest memory, and one wishes to minimize the communication costs across every level of the hierarchy. A *cache-oblivious* algorithm is one that requires no tuning based on the machine parameters M and L. An algorithm that is cache-oblivious and communication-optimal in the two-level model, such as the SMLU algorithm that is the subject of this paper, is also communication-optimal with respect to every level of any hierarchical model.

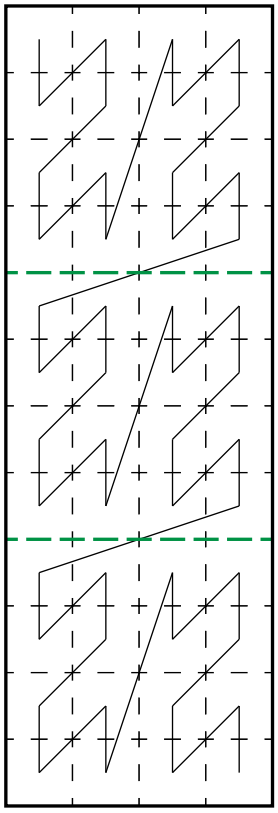
3 Data Layouts

We consider two main data layouts: *column major (CM)* and *rectangular recursive (RR)*. The CM layout is the layout used by standard libraries like LAPACK and stores each column contiguously with elements in a column ordered from top to bottom and columns themselves ordered from left to right. The RR layout is a generalization of Morton ordering [16], which is well-defined for square matrices with dimension a power of two.

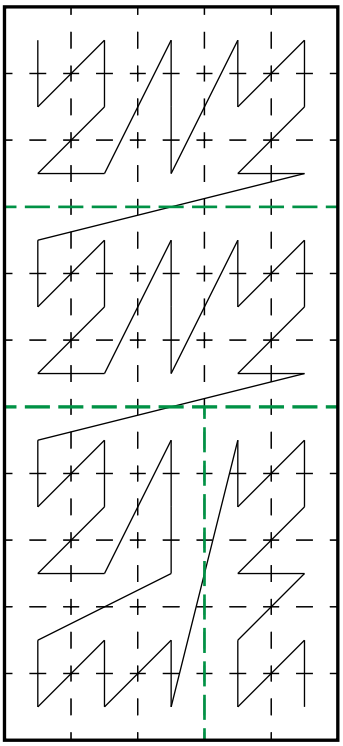
The main motivation for recursive layouts like RR is that they map well to recursive algorithms: at every node in the recursion tree, the computation involves submatrices which are stored contiguously in memory. The RR layout, illustrated in Figure 1b, corresponds to recursively splitting the largest dimension of the matrix and storing each of the two submatrices contiguously in memory. Choosing how to break ties for a square matrix (choosing whether to split horizontally or vertically) and deciding how to split odd dimensions leads to several variations of the RR layout. Here, we choose to split square matrices into left and right halves because that corresponds most closely to the CM layout, and for odd dimensions, we choose to assign the extra row to top halves and the extra column to left halves. The latter decision is arbitrary but the same choice must be made throughout the algorithm. When applied to square power-of-two matrices, our choices lead to a standard $\mathcal{U}$-Morton ordering.

There are several alternatives for generalizing Morton ordering [7, 8, 9, 12]. The simplest approach is to pad both rows and columns with zeros to obtain a square power-of-two matrix. However this can increase the number of matrix elements by a factor of 4 times the ratio of large dimension to small dimension. This approach is explored in [9], where the authors avoid the extra space and computation on padded rows and columns using "decorations" which denote full, partial, and zero submatrices. Hybrid layouts are also often used, storing small blocks in column or row-major layout and ordering the blocks using a Morton ordering. One can view our RR layout as the "recursive block column layout" from [7] with 1×1 block sizes.

We consider another alternative for generalizing Morton ordering to a specific class of rectangular matrices. If the smaller dimension of a rectangular matrix is a power of two and the larger dimension is a multiple of the smaller dimension, then the matrix can be divided up into several square power-of-two matrices. In this case, the elements within the square submatrices can be stored in standard Morton ordering, and the squares themselves can be ordered from top to bottom or left to right. This layout is illustrated in Figure 1a. For the purposes of LU and QR factorizations, if the original matrix is square with power of two dimension, then all submatrices encountered can be stored in this layout. To preserve generality and avoid padding the original matrix, we describe our algorithms with the RR layout instead of this "stack of squares" layout.



(a) Stacks of squares layout for a 12×4 matrix. The 3 square 4×4 blocks are stored contiguously, each in Morton order.



(b) Rectangular recursive (RR) order for a 11×5 matrix. At the first level, the top 6 rows are split from the bottom 5. At the second level, the top 6×5 block is split into two 3×5 blocks, whereas the bottom 5×5 block is split into a 5×3 block and a 5×2 block.

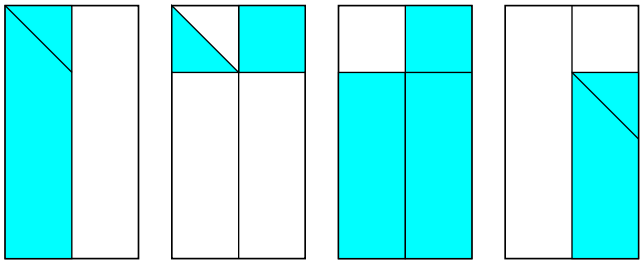


Figure 1: Cartoon of rectangular recursive algorithm for LU [17]. Shaded areas correspond to computation. In SMLU, the first and fourth steps assume column-major ordering, and the second and third steps assume rectangular recursive ordering.

4 Rectangular Recursive Algorithms for LU and QR

Many recursive algorithms for linear algebra computations are cache-oblivious, but in order to minimize latency costs the data layout must be chosen carefully. Morton ordering works very well for the recursive matrix multiplication algorithm, where the eight recursive subproblems involve matrix quadrants. The natural extension of Morton ordering to symmetric matrices also maps nicely to the square recursive algorithm for Cholesky decomposition [1, 2, 13]. In this algorithm, subroutines and recursive subproblems involve matrix quadrants (which may be symmetric, triangular, or dense).

For LU decomposition, the analogous square recursive algorithm (and standard Morton ordering) is not sufficient. In order to maintain numerical stability, row (and possibly column) interchanges are necessary. Partial pivoting, the most common scheme, involves at each step of the algorithm selecting the maximum element in absolute value in a column and interchanging the corresponding row with the diagonal element's row. For this reason, the square recursive algorithm for Cholesky does not generalize to nonsymmetric matrices: the top left quadrant of the matrix cannot be factored without accessing (and possibly interchanging) rows from the bottom left quadrant of the matrix.

In order to respect the column-access requirement of partial pivoting, Toledo [17] and Gustavson [13] developed a "rectangular recursive" algorithm which recursively splits the matrix into left and right halves instead of quadrants. The steps of the computation are shown in Figure 1. Given an $m \times n$ input matrix, recursive subproblems are of size $m \times \frac{n}{2}$ and $\left(m - \frac{n}{2}\right) \times \frac{n}{2}$, and algorithms for triangular solve with multiple right hand sides (TRSM) and matrix multiplication are used as subroutines. Because the recursion splits the matrix into left and right halves, the base of the recursion consists of factoring single columns with partial pivoting: finding the maximum element, swapping it with the diagonal, and scaling the column with its reciprocal.

A similar algorithm for QR decomposition was developed by Elmroth and Gustavson [6]. The standard Householder QR algorithm works column-by-column, computing a Householder vector that annihilates all subdiagonal entries in the column and applying the orthogonal transformation to the trailing matrix. In order to compute one Householder vector per column, a rectangular recursive algorithm is necessary so that the base of the recursion consists of computing a single Householder vector to annihilate the entire column below the diagonal. The basic steps of the computation are shown in Figure 2. In the rectangular recursive QR algorithm, an auxiliary triangular matrix T is computed so that the update of the trailing matrix can be done with matrix multiplication.

Abandoning the requirement that the orthogonal factor Q be computed with one Householder vector per column allows for a square recursive algorithm for QR [9]. The square recursive algorithm maps nicely onto standard Morton ordering, as each computation involves matrix quadrants. However, because the orthogonalization is based on many Givens rotations per column instead of one Householder vector per

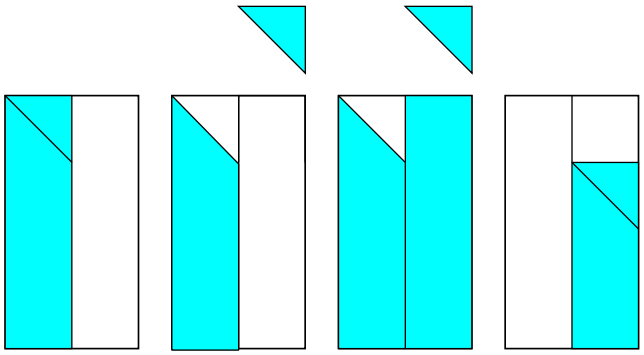


Figure 2: Cartoon of rectangular recursive algorithm for QR [6]. Shaded areas correspond to computation. The triangles correspond to the intermediate T factor. In SMQR, the first and fourth steps assume column-major ordering, and the second and third steps assume rectangular recursive ordering.

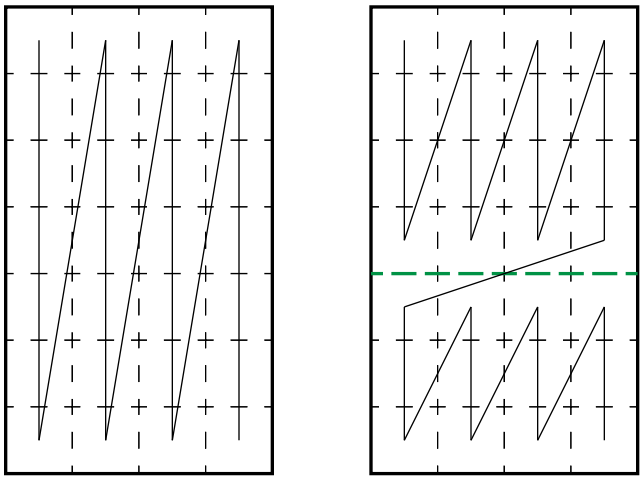


Figure 3: One recursive step in converting from column-major to rectangular recursive order.

column, the standard trailing matrix update techniques do not apply. The approach from [9] is to explicitly construct the orthogonal factor Q, using matrix multiplication to update the trailing matrix. This technique leads to an increase in the total flop count of the decomposition compared to the standard algorithm, by a factor of approximately $3\times$.

By using shape morphing, we show that the rectangular recursive algorithm of Elmroth and Gustavson [6] can maintain the standard format of representing the orthogonal factor by its Householder vectors (one per column) and still achieve cache-obliviousness, minimizing both words and messages. The rectangular recursive algorithm also increases the flop count with respect to the standard algorithm, by about 17% for tall skinny matrices and about 30% for square matrices. To limit the increase in computation, one can use a hybrid algorithm, using the rectangular recursive algorithm on panels of sufficiently small width. Since this tuning parameter prevents the algorithm from begin cache-oblivious, we do not consider the hybrid algorithm here.

5 Subroutines and Their Communication Costs

5.1 Converting Rectangular Recursive to Column Major and Back

The algorithm for reshaping a matrix from column-major order to rectangular recursive order is provided in Algorithm 4. The algorithm is recursive; at each step it splits the matrix along its largest dimension and is then recursively called on both submatrices. When the input is short and fat ($m \leq n$), splitting the matrix does not require any data movement, since in column-major order the left and right halves of the matrix are already contiguous. When the input is tall and skinny ($m > n$), splitting the matrix requires "separating" each column into its top and bottom halves. We perform this operation with the Separate function: since it involves contiguously streaming through the input and contiguously writing to two output locations, the communication cost is $O(mn)$ words and $O\left(\frac{mn}{L}\right)$ messages, as illustrated in Figure 3. The recurrence for the communication cost is therefore

$$\text{Reshape}(n,m) = \begin{cases} 2\text{Reshape}(m/2,n) + O((mn/L)\alpha + mn\beta) & m > n \text{ and } mn > M \\ 2\text{Reshape}(m,n/2) & m \leq n \text{ and } mn > M \\ O((mn/L+1)\alpha + mn\beta) & mn \leq M \end{cases},$$

with solution

$$\text{Reshape}(n,m) = O\left(\left(\frac{mn}{L}\log\frac{mn}{M}+1\right)\alpha + mn\left(\log\frac{mn}{M}+1\right)\beta\right).$$

Reshaping from rectangular recursive order to column-major order is described in Algorithm 10, and has identical costs.

Algorithm 3 $\begin{pmatrix} A_1 \\ A_2 \end{pmatrix} = \text{Separate}(A,m,n,m_1)$

Input: A is $m \times n$ in column-major order
Output: A_1 is the first m_1 rows of A in column-major order, A_2 is the remaining $m - m_1$ rows of A in column-major order
  for j in 1:n **do**
    $A_1(1:m_1, j) = A(1:m_1, j)$
    $A_2(1:m-m_1, j) = A(m_1+1:m, j)$
  end for

Algorithm 4 $B = \text{ReshapeToRecursive}(A,m,n)$

Input: A is $m \times n$ with $m \geq n$ in column-major order
Output: B is the same matrix in rectangular recursive order
  if $m = n = 1$ **then**
    $B(1,1) = A(1,1)$
    return
  end if
  if $m > n$ **then**
    $m_1 = \lceil m/2 \rceil, m_2 = \lfloor m/2 \rfloor$
    $\begin{pmatrix} B_1 \\ B_2 \end{pmatrix} = \text{Separate}(A,m,n,m_1)$
    $B_1 = \text{ReshapeToRecursive}(B_1,m_1,n)$
    $B_2 = \text{ReshapeToRecursive}(B_2,m_2,n)$
    $B = \begin{pmatrix} B_1 \\ B_2 \end{pmatrix}$
  else
    $n_1 = \lceil n/2 \rceil, n_2 = \lfloor n/2 \rfloor$
    $\begin{pmatrix} B_1 & B_2 \end{pmatrix} = A$
    $B_1 = \text{ReshapeToRecursive}(B_1,m,n_1)$
    $B_2 = \text{ReshapeToRecursive}(B_2,m,n_2)$
    $B = \begin{pmatrix} B_1 & B_2 \end{pmatrix}$
  end if

5.2 Rectangular Matrix Multiplication

The SMLU algorithm requires a recursive matrix multiplication algorithm for square matrices stored in rectangular recorsive order. The communication costs of recursive matrix multiplication were first analyzed in [10]. In our model they are worked out in [2] and are

$$\text{GEMM}(m,n,k) = O\left(\left(\frac{mnk}{\sqrt{M}L} + \frac{mn+mk+nk}{L} + 1\right)\alpha + \left(\frac{mnk}{\sqrt{M}} + mn + mk + nk\right)\beta\right),$$

where m, n, k are the three matrix dimensions. For completeness, the algorithm for rectangular recursive layout appears in Algorithm 11.

5.3 Rectangular Triangular Solve

The SMLU algorithm requires a recursive triangular solve on matrices stored in Morton order. An algorithm for square matrices with optimal communication costs is given in [2]. In Algorithm 5 we generalize to the case of rectangular matrices. Let A be an $m \times n$ matrix, and L be a $m \times m$ unit lower triangular matrix.[1] At each recursive step, split the larger of m and n. Splitting m gives two recursive calls to TRSM and one call to matrix multiplication. Splitting n gives two recursive calls to TRSM. Thus the communication cost recurrence is:

$$\text{TRSM}(m,n) = \begin{cases} 2\text{TRSM}(m/2,n) + \text{GEMM}(m,m,n) & m > n \quad \text{and} \quad 2mn + m^2 > M \\ 2\text{TRSM}(m,n/2) & n \geq m \quad \text{and} \quad 2mn + m^2 > M \\ O\left(((mn+m^2)/L + 1)\alpha + (mn+m^2)\beta\right) & 2mn + m^2 \leq M \end{cases},$$

with solution

$$\text{TRSM}(m,n) = O\left(\left(\frac{m^2 n}{L\sqrt{M}} + \frac{mn+m^2}{L} + 1\right)\alpha + \left(\frac{m^2 n}{\sqrt{M}} + mn + m^2\right)\beta\right).$$

5.4 Pivoting

The SMLU algorithm returns a pivot vector p of length m, where $p(i) = j$ indicates that row j in the original matrix has been pivoted to row i in the output. Two subroutines are required to manage the pivoting.

First, ApplyPivots, presented as Algorithm 6, applies a pivot vector to a matrix. It applies the pivot vector to each column of the matrix in sequence. For each column, it applies the pivot vector recursively by streaming through the entire column to separate entries between those that belong in the the top half from those that belong in the bottom half of the permuted column, the calling itself on both the top and bottom halves. If $m < M$, at least one column fits into memory and ApplyPivots needs to read the matrix only once. If $m > M$, it reads and writes each column $\log(m/M)$ times. The communication costs are

$$\text{ApplyPivots}(m,n) = O\left(\left(\frac{mn}{L}\left(1 + \log\frac{m}{M}\right) + 1\right)\alpha + mn\left(1 + \log\frac{m}{M}\right)\beta\right).$$

It is also necessary to combine two pivot vectors into one, which is done by CombinePivots, presented in Algorithm 7. This is accomplished by two calls to ApplyPivots with $n = 1$, so the communication costs are

$$\text{CombinePivots}(m) = O\left(\left(\frac{m}{L}\left(1 + \log\frac{m}{M}\right) + 1\right)\alpha + m\left(1 + \log\frac{m}{M}\right)\beta\right).$$

[1] A non-unit lower triangular matrix changes only the base case computation.

Algorithm 5 $U = \text{RecTRSM}(A,L,m,n)$

Input: A is $m \times n$, L is $m \times m$ and unit lower triangular, both in rectangular recursive layout
Output: $U = L^{-1}A$ in rectangular recursive layout

if $m = n = 1$ **then**
  $U(1,1) = A(1,1)$
  return
end if
if $m > n$ **then**
  $m_1 = \lceil m/2 \rceil$, $m_2 = \lfloor m/2 \rfloor$
  $\begin{pmatrix} L_{11} & \\ L_{21} & L_{22} \end{pmatrix} = L$
  $\begin{pmatrix} U_1 \\ U_2 \end{pmatrix} = U$
  $U_1 = \text{RecTRSM}(U_1, L_{11}, m_1, n)$
  $U_2 = \text{RecGEMM}(L_{21}, U_1, U_2, m_2, m_1, n)$
  $U_2 = \text{RecTRSM}(U_2, L_{22}, m_2, n)$
  $U = \begin{pmatrix} U_1 \\ U_2 \end{pmatrix}$
else
  $n_1 = \lceil n/2 \rceil$, $n_2 = \lfloor n/2 \rfloor$
  $\begin{pmatrix} U_1 & U_2 \end{pmatrix} = U$
  $U_1 = \text{RecTRSM}(U_1, L, m, n_1)$
  $U_2 = \text{RecTRSM}(U_2, L, m, n_2)$
  $U = \begin{pmatrix} U_1 & U_2 \end{pmatrix}$
end if

Algorithm 6 $\text{ApplyPivots}(A,P,m,n)$

Input: A is $m \times n$ in column-major order, P is a pivot vector
Output: The rows of A are pivoted according to P

if $m = n = 1$ **then**
  return
end if
if $n = 1$ **then**
  $m_1 = \lceil m/2 \rceil$, $m_2 = \lfloor m/2 \rfloor$
  $c_1 =$ new array of length m_1
  $c_2 =$ new array of length m_2
  $P_1 =$ new array of length m_1
  $P_2 =$ new array of length m_2
  $j = 1$; $k = 1$
  for i in $1 : n$ **do**
    if then $P(i) \leq m_1$
      $c_1(j) = A(i)$
      $P_1(j) = P(i)$ $j = j + 1$
    else
      $c_2(j) = A(i)$
      $P_2(j) = P(i) - m_1$ $k = k + 1$
    end if
  end for
  $\text{ApplyPivots}(c_1, P_1, m_1, 1)$
  $\text{ApplyPivots}(c_2, P_2, m_2, 1)$
else
  $n_1 = \lceil n/2 \rceil$, $n_2 = \lfloor n/2 \rfloor$
  $\begin{pmatrix} A_1 & A_2 \end{pmatrix} = A$
  $\text{ApplyPivots}(A_1, P, m, n_1)$
  $\text{ApplyPivots}(A_2, P, m, n_2)$
end if

Algorithm 7 $P = \text{CombinePivots}(P_L, P_R, m_L, m_R)$

Input: P_L, P_R are left and right pivot vectors
Output: P is the combined pivot vector

// Convert the size of the right pivot vector
$k = m_L - m_R$
$P'_R =$ new vector of length m_L
$P'_R(1:k) = 1:k$
$P'_R(k+1:m_L) = P_R + k$

// Combine pivots
$P_I = \text{ApplyPivots}(1:m_L, P_L, m_L, 1)$
$P = \text{ApplyPivots}(P'_R, P_I, m_L, 1)$

5.5 Analysis of SMLU

Detailed pseudocode for SMLU appears in Algorithm 8. Each call to SMLU has two recursive calls to itself, two calls to ApplyPivots, four calls each to ReshapeToRecursive and ReshapeToColMajor, one call to RecTRSM, one to RecGEMM, and one call to CombinePivots. The recursive communication costs are thus

$$\begin{aligned}\text{SMLU}(m,n) &\le 2\text{SMLU}\left(m, \frac{n}{2}\right) + 2\text{ApplyPivots}\left(m, \frac{n}{2}\right) + 8\text{Reshape}\left(m, \frac{n}{2}\right) + \text{TRSM}\left(\frac{n}{2}, \frac{n}{2}\right) + \\ &\quad + \text{GEMM}\left(m, \frac{n}{2}, \frac{n}{2}\right) + \text{CombinePivots}(m) \\ &= 2\text{SMLU}\left(m, \frac{n}{2}\right) + O\left(\left(\frac{mn^2}{\sqrt{M}} + mn\log\frac{mn}{M} + mn\right)\left(\beta + \frac{\alpha}{L}\right)\right).\end{aligned}$$

If $M < m$, one column of the matrix does not fit in fast memory, so the base case costs are $\text{SMLU}(1,m) = m\left(\beta + \frac{\alpha}{L}\right)$. If $M \ge m$, then M/m columns fit into fast memory at once, so the base case costs are $\text{SMLU}\left(\frac{M}{m}, m\right) = M\left(\beta + \frac{\alpha}{L}\right)$. The solution to the recurrence is

$$\text{SMLU}(m,n) = \begin{cases} O\left(\left(\frac{mn^2}{M^{1/2}} + mn\log\frac{mn}{M}\log n\right)\left(\beta + \frac{\alpha}{L}\right) + \alpha\right) & M < m \\ O\left(\left(\frac{mn^2}{M^{1/2}} + mn\left(\log\frac{mn}{M}\right)^2 + mn\right)\left(\beta + \frac{\alpha}{L}\right) + \alpha\right) & M \ge m \end{cases}$$

Recall that the communication lower bound for LU [3], which is attainable for LU without pivoting, is

$$\text{LU}(m,n) = \Omega\left(\left(\frac{mn^2}{M^{1/2}} + mn\right)\left(\beta + \frac{\alpha}{L}\right) + \alpha\right).$$

Compared to this lower bound, SMLU has an extra polylogarithmic factor on the mn term. In the square case, $m = n$, SMLU asymptotically matches the lower bound except in the tiny range

$$\frac{n^2}{(\log(mn))^4} < M < n^2.$$

In the rectangular case, SMLU may be larger than the lower bound by a logarithmic factor in a larger range

$$\frac{n^2}{(\log(mn))^4} < M < mn.$$

Compared to the original rectangular recursive algorithm for LU [13, 17], with partial pivoting but without shape morphing, SMLU has a communication cost with an extra $\log(mn/M)$ on the mn term. Thus, outside the ranges given above, shape morphing does not increase the communication costs asymptotically.

Algorithm 8 $P = \text{SMLU}(A, m, n)$

if n = 1 **then**
$P = 1 : m$
$i = \text{ArgMax}(|A|)$
Swap$(A(1), A(i))$
Swap$(P(1), P(i))$
Scale$(A(2 : m), 1/A(1))$
else

// set submatrix dimensions
$n_1 = \lceil \frac{n}{2} \rceil$
$n_2 = n - n_1$
$m_1 = n_1$
$m_2 = m - m_1$

// recurse on left half
$\begin{pmatrix} A_1 & A_2 \end{pmatrix} = A$
$P_{\text{L}} = \text{SMLU}(A_1, m, n_1)$

// forward pivot
ApplyPivots$(A_2, P_{\text{L}}, m, n_2)$

// separate top m_1 rows from bottom m_2 rows
$\begin{pmatrix} A_{11} \\ A_{21} \end{pmatrix} = \text{Separate}(A_1, m, n_1, m_1)$
$\begin{pmatrix} A_{12} \\ A_{22} \end{pmatrix} = \text{Separate}(A_2, m, n_2, m_1)$

// convert each quadrant to Morton ordering
ReshapeToRecursive(A_{11}, m_1, n_1)
ReshapeToRecursive(A_{12}, m_1, n_2)
ReshapeToRecursive(A_{21}, m_2, n_1)
ReshapeToRecursive(A_{22}, m_2, n_2)

// triangular solve with Morton ordered arrays
$A_{12} = \text{RecTRSM}(A_{12}, A_{11}, n_1, n_2)$

// Schur update with Morton ordered arrays
$A_{22} = \text{RecGEMM}(A_{21}, A_{12}, A_{22}, m_2, n_1, n_2)$

// convert quadrants back to column major
$A_{11} = \text{ReshapeToColMajor}(A_{11}, m_1, n_1)$
$A_{12} = \text{ReshapeToColMajor}(A_{12}, m_1, n_2)$
$A_{21} = \text{ReshapeToColMajor}(A_{21}, m_2, n_1)$
$A_{22} = \text{ReshapeToColMajor}(A_{22}, m_2, n_2)$

// recurse on (bottom of) right half
$P_{\text{R}} = \text{SMLU}(A_{22}, m_2, n_2)$

// back pivot
ApplyPivots$(A_{21}, P_{\text{R}}, m_2, n_1)$

// combine pivots
$P = \text{CombinePivots}(P_{\text{L}}, P_{\text{R}}, m, m_2)$
$A = \text{Combine}(\begin{pmatrix} A_{11} & A_{12} \end{pmatrix}, \begin{pmatrix} A_{21} & A_{22} \end{pmatrix}, m, n, m_1)$
end if

References

[1] N. Ahmed and K. Pingali. Automatic generation of block-recursive codes. In *Euro-Par '00: Proceedings from the 6th International Euro-Par Conference on Parallel Processing*, pages 368–378, London, UK, 2000. Springer-Verlag.

[2] G. Ballard, J. Demmel, O. Holtz, and O. Schwartz. Communication-optimal parallel and sequential Cholesky decomposition. In *SPAA '09: Proceedings of the Twenty-First Annual Symposium on Parallelism in Algorithms and Architectures*, pages 245–252, New York, NY, USA, 2009. ACM.

[3] G. Ballard, J. Demmel, O. Holtz, and O. Schwartz. Minimizing communication in numerical linear algebra. *SIAM J. Matrix Analysis Applications*, 32(3):866–901, 2011.

[4] J. Demmel, I. Dumitriu, and O. Holtz. Fast linear algebra is stable. *Numerische Mathematik*, 108(1):59–91, 2007.

[5] J. W. Demmel. LAPACK Working Note 53: Trading off parallelism and numerical stability. Technical report, Knoxville, TN, USA, 1992.

[6] E. Elmroth and F. Gustavson. New serial and parallel recursive QR factorization algorithms for SMP systems. *Applied Parallel Computing Large Scale Scientific and Industrial Problems*, pages 120–128, 1998.

[7] E. Elmroth, F. Gustavson, I. Jonsson, and B. Kågström. Recursive blocked algorithms and hybrid data structures for dense matrix library software. *SIAM Review*, 46(1):3–45, 2004.

[8] J. Frens and D. S. Wise. Auto-blocking matrix-multiplication or tracking BLAS3 performance from source code. In *In Proceedings of the Sixth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 206–216, 1997.

[9] J. D. Frens and D. S. Wise. QR factorization with Morton-ordered quadtree matrices for memory re-use and parallelism. *SIGPLAN Not.*, 38(10):144–154, 2003.

[10] M. Frigo, C. E. Leiserson, H. Prokop, and S. Ramachandran. Cache-oblivious algorithms. In *FOCS '99: Proceedings of the 40th Annual Symposium on Foundations of Computer Science*, pages 285–297, Washington, DC, USA, 1999. IEEE Computer Society.

[11] L. Grigori, J. Demmel, and H. Xiang. CALU: A communication optimal LU factorization algorithm. *SIAM Journal on Matrix Analysis and Applications*, 32(4):1317–1350, 2011.

[12] F. Gustavson, A. Henriksson, I. Jonsson, B. Kågström, and P. Ling. Recursive blocked data formats and BLAS's for dense linear algebra algorithms. *Applied Parallel Computing Large Scale Scientific and Industrial Problems*, pages 195–206, 1998.

[13] F. G. Gustavson. Recursion leads to automatic variable blocking for dense linear-algebra algorithms. *IBM J. Res. Dev.*, 41(6):737–756, 1997.

[14] D. Irony, S. Toledo, and A. Tiskin. Communication lower bounds for distributed-memory matrix multiplication. *J. Parallel Distrib. Comput.*, 64(9):1017–1026, 2004.

[15] A. Khabou, J. Demmel, L. Grigori, and M. Gu. LU factorization with panel rank revealing pivoting and its communication avoiding version. Technical Report UCB/EECS-2012-15, EECS Department, University of California, Berkeley, Jan 2012.

[16] G. M. Morton. *A Computer Oriented Geodetic Data Base and a New Technique in File Sequencing*. International Business Machines Company, 1966.

[17] S. Toledo. Locality of reference in LU decomposition with partial pivoting. *SIAM J. Matrix Anal. Appl.*, 18(4):1065–1081, 1997.

A Supplementary Algorithms

Algorithm 9 $A = \text{Combine}(A_1, A_2, m, n, m_1)$

Input: A_1 is $m_1 \times n$ and A_2 is $m - m_1 \times n$ both in column-major order
Output: A is $m \times n$, the first m_1 rows are from A_1 and the remaining rows are from A_2
for j in 1:n **do**
$A(1 : m_1, j) = A_1(1 : m_1, j)$
$A(m_1 + 1 : m, j) = A_2(1 : m - m_1, j)$
end for

Algorithm 10 $B = \text{ReshapeToColMajor}(A, n, m)$

Input: A is $m \times n$ with $m \geq n$ in rectangular recursive order
Output: B is the same matrix in column-major order
if $m = n = 1$ **then**
$B(1, 1) = A(1, 1)$
return
end if
if $m > n$ **then**
$m_1 = \lceil m/2 \rceil$, $m_2 = \lfloor m/2 \rfloor$
$\begin{pmatrix} B_1 \\ B_2 \end{pmatrix} = A$
$B_1 = \text{ReshapeToColMajor}(B_1, m_1, n)$
$B_2 = \text{ReshapeToColMajor}(B_2, m_2, n)$
$B = \text{Combine}(B_1, B_2, m, n, m_1)$
else
$n_1 = \lceil n/2 \rceil$, $n_2 = \lfloor n/2 \rfloor$
$\begin{pmatrix} B_1 & B_2 \end{pmatrix} = A$
$B_1 = \text{ReshapeToColMajor}(B_1, m, n_1)$
$B_2 = \text{ReshapeToColMajor}(B_2, m, n_2)$
$B = \begin{pmatrix} B_1 & B_2 \end{pmatrix}$
end if

Algorithm 11 $C = \text{RecGEMM}(A,B,C,m,k,n)$

Input: A is $m \times k$, B is $k \times n$, C is $m \times n$, all in rectangular recursive layout
Output: $C = C - A \cdot B$

if $m = 0$ or $n = 0$ or $k = 0$ **then**
  return
end if
if $m = n = k = 1$ **then**
  $C(1,1) = C(1,1) - A(1,1) \cdot B(1,1)$
  return
end if
if $k \geq m$ and $k > n$ **then**
  $k_1 = \lceil k/2 \rceil$, $k_2 = \lfloor k/2 \rfloor$
  $\begin{pmatrix} A_1 & A_2 \end{pmatrix} = A$
  $\begin{pmatrix} B_1 \\ B_2 \end{pmatrix} = B$
  RecGEMM(A_1,B_1,C,m,n,k_1)
  RecGEMM(A_2,B_2,C,m,n,k_2)
end if
if $m > k$ and $m > n$ **then**
  $m_1 = \lceil m/2 \rceil$, $m_2 = \lfloor m/2 \rfloor$
  $\begin{pmatrix} A_1 \\ A_2 \end{pmatrix} = A$
  $\begin{pmatrix} C_1 \\ C_2 \end{pmatrix} = C$
  RecGEMM(A_1,B,C_1,m_1,n,k)
  RecGEMM(A_2,B,C_2,m_2,n,k)
  $C = \begin{pmatrix} C_1 \\ C_2 \end{pmatrix}$
end if
if $n \geq k$ and $n \geq m$ **then**
  $n_1 = \lceil n/2 \rceil$, $n_2 = \lfloor n/2 \rfloor$
  $\begin{pmatrix} B_1 & B_2 \end{pmatrix} = B$
  $\begin{pmatrix} C_1 & C_2 \end{pmatrix} = C$
  RecGEMM(A,B_1,C_1,m,n_1,k)
  RecGEMM(A,B_1,C_2,m,n_1,k)
  $C = \begin{pmatrix} C_1 & C_2 \end{pmatrix}$
end if